# Laboratorium 5 Testy jednostowe

Autor: Krzysztof Hardek

## Zad 1

Zmiana naliczanego podatku z 22% na 23%.

### Klasy funkconalne

**Klasa Order**

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

### Klasy testowe

**Klasa OrderTest**

Zmiana oczekiwanej wartości na poprawną.

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

## Zad 2

Więcej produktow w jednym zamówieniu

### Klasy funkcjonalne

**Klasa Order**

Zmina pola product na products

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

Modyfikacja konstruktora.

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

Uaktualnienie getera.

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

getPrice

```
public BigDecimal getPrice() {
    BigDecimal totalPrice = BigDecimal.valueOf(0);

    for(Product product: products){
        totalPrice.add(product.getPrice());
    }

    return totalPrice;
}
```

## Klasy testowe

### Klasa OrderTest

Nadpisałem odpowiendnie metody, tak aby korzystały z listy produktów.

```
private Order getOrderWithMockedProduct() {
    List<Product> products = new ArrayList<>();
    products.add(mock(Product.class));

    return new Order(products);
}

@Test
public void testGetProductThroughOrder() {
    // given
    List<Product> expectedProducts = new ArrayList<>();
    expectedProducts.add(mock(Product.class));

    Order order = new Order(expectedProducts);

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

    // then
    assertSame(expectedProducts, actualProducts);
}

public void testGetPrice() throws Exception {
```

```
        // given
        BigDecimal expectedProductPrice = BigDecimal.valueOf(1000);
        Product product = mock(Product.class);
        given(product.getPrice()).willReturn(expectedProductPrice);

        List<Product> products = new ArrayList<>();
        products.add(product);

        Order order = new Order(products);

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

        // then
        assertBigDecimalCompareValue(expectedProductPrice, actualProductPrice);
    }

    private Order getOrderWithCertainProductPrice(double productPriceValue) {
        BigDecimal productPrice = BigDecimal.valueOf(productPriceValue);
        Product product = mock(Product.class);

        List<Product> products = new ArrayList<>();
        products.add(product);

        given(product.getPrice()).willReturn(productPrice);
        return new Order(products);
    }
```

# Zad 3

Możliwość naliczania rabatu dla pojedyńczego produktu oraz całego zamówienia.

## Klasy funkcjonalne

### Klasa Order

Dodałem pole discount.

```
private double discount;
```

Uaktualniłem konstruktor.

```
public Order(List<Product> products) {
    if(products == null){
        throw new IllegalArgumentException();
    }

    this.products = products;
    id = UUID.randomUUID();
```

```
    paid = false;
    this.discount = 0;
}
```

Dodałem geter i seter na to pole.

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

public void setDiscount(double discount) {
    if(discount > 1 || discount < 0){
        throw new IllegalArgumentException();
    }

    this.discount = discount;
}
```

Zmieniłem getPrice.

```
public BigDecimal getPrice() {
    BigDecimal totalPrice = new BigDecimal(0);

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

    return totalPrice.multiply(new BigDecimal(1-this.discount));
}
```

## Klasa Product

Analogicznie w tej klasie.

```
private double discount;

public Product(String name, BigDecimal price) {
    this.name = name;
    this.price = price;
    this.price.setScale(PRICE_PRECISION, ROUND_STRATEGY);
    this.discount = 0;
}

public BigDecimal getPrice() {
    return price.multiply(new BigDecimal(1-this.discount));
}
```

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

public void setDiscount(double discount) {
    if(discount > 1 || discount < 0){
        throw new IllegalArgumentException();
    }

    this.discount = discount;
}
```

# Zad 4

## Klasy funkcjonalne

### Klasa OrderHistory

Przechowuje historie zamówień oraz umożliwia wyszukiwanie.

```
public class OrderHistory {
    private List<Order> orders;

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

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

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

    public List<Order> searchOrders(SearchStrategy strategy){
        return this.orders.stream().filter(order ->
strategy.filter(order)).collect(Collectors.toList());
    }
}
```

### Interfejs SearchStrategy

Element Composite Pattern.

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

**Klasa CompositeSearchStrategy**

Element Composite Pattern. Grupuje wyszukania.

```
public class CompositeSearchStrategy implements SearchStrategy{
    private final List<SearchStrategy> strategies;

    public CompositeSearchStrategy(List<SearchStrategy> strategies){
        this.strategies = strategies;
    }

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

**Klasa ProductNameSearchStrategy**

Wyszukiwanie po nazwie produktu.

```
public class ProductNameSearchStrategy implements SearchStrategy{
    private String name;

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

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

**Klasa OrderPriceSearchStrategy**

Wyszukiwanie po cenie zamówienia.

```
public class OrderPriceSearchStrategy implements SearchStrategy{
    private BigDecimal minPrice;
    private BigDecimal maxPrice;

    public OrderPriceSearchStrategy(BigDecimal minPrice, BigDecimal
maxPrice){
```

```
        this.maxPrice = maxPrice;
        this.minPrice = minPrice;
    }

    @Override
    public boolean filter(Order order) {
        return order.getPriceWithTaxes().compareTo(minPrice) >= 0 &&
order.getPriceWithTaxes().compareTo(maxPrice) <= 0;
    }
}
```

**Klasa ClientNameSearchStrategy**

Wyszukiwanie po Nazwie klienta.

```
public class ClientNameSearchStrategy implements SearchStrategy{
    private String name;

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

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

## Klasy testowe

**Klasa OrderHistoryTest**

```
public class OrderHistoryTest {
    private Order getOrderWithCertainPriceAndNames(BigDecimal price, String
clientName, String productName){
        Address address = mock(Address.class);
        given(address.getName()).willReturn(clientName);
        Shipment shipment = mock(Shipment.class);
        given(shipment.getRecipientAddress()).willReturn(address);
        List<Product> products = new ArrayList<>();
        Product product = mock(Product.class);
        given(product.getName()).willReturn(productName);
        products.add(product);
        Order order = mock(Order.class);
        given(order.getShipment()).willReturn(shipment);
        given(order.getPriceWithTaxes()).willReturn(price);
        given(order.getProducts()).willReturn(products);
```

```
        return order;
    }

    @Test
    public void searchTest(){
        //given
        OrderPriceSearchStrategy orderPriceSearchStrategy = new
OrderPriceSearchStrategy(
                new BigDecimal(1), new BigDecimal(7));

        ClientNameSearchStrategy clientNameSearchStrategy = new
ClientNameSearchStrategy(
                "Hardek");
        ProductNameSearchStrategy productNameSearchStrategy = new
ProductNameSearchStrategy(
                "Marchewka");

        List<SearchStrategy> searchStrategies = new ArrayList<>();
        searchStrategies.add(orderPriceSearchStrategy);
        searchStrategies.add(clientNameSearchStrategy);
        searchStrategies.add(productNameSearchStrategy);

        CompositeSearchStrategy compositeSearchStrategy = new
CompositeSearchStrategy(searchStrategies);

        Order order1 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Hardek", "Marchewka");
        Order order2 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Nowak", "Marchewka");
        Order order3 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Nowak", "Marchewka");
        Order order4 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Hardek", "Marchewka");
        Order order5 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Hardek", "Ogorek");
        Order order6 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Nowak", "Ogorek");
        Order order7 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Nowak", "Ogorek");
        Order order8 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Hardek", "Ogorek");

        //when
        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);
        orderHistory.addOrder(order3);
        orderHistory.addOrder(order4);
        orderHistory.addOrder(order5);
        orderHistory.addOrder(order6);
        orderHistory.addOrder(order7);
        orderHistory.addOrder(order8);

        List<Order> expectedOrders = new ArrayList<>();
```

```
        expectedOrders.add(order1);
        expectedOrders.add(order4);

        //then
        assertEquals(expectedOrders,
orderHistory.searchOrders(compositeSearchStrategy));
    }

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

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

        //when
        List<Order> expectedOrders = new ArrayList<>();
        expectedOrders.add(order1);
        expectedOrders.add(order2);

        //then
        assertEquals(expectedOrders, orderHistory.getOrders());
    }
}
```

**Klasa CompositeSearchStrategyTest**

```
public class CompositeSearchStrategyTest {
    private Order getOrderWithCertainPriceAndNames(BigDecimal price, String
clientName, String productName){
        Address address = mock(Address.class);
        given(address.getName()).willReturn(clientName);
        Shipment shipment = mock(Shipment.class);
        given(shipment.getRecipientAddress()).willReturn(address);
        List<Product> products = new ArrayList<>();
        Product product = mock(Product.class);
        given(product.getName()).willReturn(productName);
        products.add(product);
        Order order = mock(Order.class);
        given(order.getShipment()).willReturn(shipment);
        given(order.getPriceWithTaxes()).willReturn(price);
        given(order.getProducts()).willReturn(products);
        return order;
    }

    @Test
    void testSearch(){
        //given
```

```
        OrderPriceSearchStrategy orderPriceSearchStrategy = new
OrderPriceSearchStrategy(
                new BigDecimal(1), new BigDecimal(5));

        ClientNameSearchStrategy clientNameSearchStrategy = new
ClientNameSearchStrategy(
                "Hardek");
        ProductNameSearchStrategy productNameSearchStrategy = new
ProductNameSearchStrategy(
                "Marchewka");

        List<SearchStrategy> searchStrategies = new ArrayList<>();
        searchStrategies.add(orderPriceSearchStrategy);
        searchStrategies.add(clientNameSearchStrategy);
        searchStrategies.add(productNameSearchStrategy);

        CompositeSearchStrategy compositeSearchStrategy = new
CompositeSearchStrategy(searchStrategies);

        // when
        Order order1 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Hardek", "Marchewka");
        Order order2 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Nowak", "Marchewka");
        Order order3 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Nowak", "Marchewka");
        Order order4 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Hardek", "Marchewka");
        Order order5 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Hardek", "Ogorek");
        Order order6 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Nowak", "Ogorek");
        Order order7 = getOrderWithCertainPriceAndNames(new BigDecimal(4),
"Nowak", "Ogorek");
        Order order8 = getOrderWithCertainPriceAndNames(new BigDecimal(6),
"Hardek", "Ogorek");

        //then
        assertTrue(compositeSearchStrategy.filter(order1));
        assertFalse(compositeSearchStrategy.filter(order2));
        assertFalse(compositeSearchStrategy.filter(order3));
        assertFalse(compositeSearchStrategy.filter(order4));
        assertFalse(compositeSearchStrategy.filter(order5));
        assertFalse(compositeSearchStrategy.filter(order6));
        assertFalse(compositeSearchStrategy.filter(order7));
        assertFalse(compositeSearchStrategy.filter(order8));
    }
}
```

**Klasa ClientNameSearchStrategyTest**

```
public class ClientNameSearchStrategyTest {

    private Order getOrderWithCertainClientName(String name){
        Address address = mock(Address.class);
        given(address.getName()).willReturn(name);
        Shipment shipment = mock(Shipment.class);
        given(shipment.getRecipientAddress()).willReturn(address);
        Order order = mock(Order.class);
        given(order.getShipment()).willReturn(shipment);
        return order;
    }

    @Test
    void testSearch(){
        //given
        ClientNameSearchStrategy clientNameSearchStrategy = new
ClientNameSearchStrategy("Hardek");

        // when
        Order order1 = getOrderWithCertainClientName("Hardek");
        Order order2 = getOrderWithCertainClientName("Nowak");

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

**Klasa OrderPriceSearchStrategyTest**

```
public class OrderPriceSearchStrategyTest {
    private Order getOrderWithCertainPrice(BigDecimal price){
        Order order = mock(Order.class);
        given(order.getPriceWithTaxes()).willReturn(price);
        return order;
    }

    @Test
    void testSearch(){
        //given
        OrderPriceSearchStrategy orderPriceSearchStrategy = new
OrderPriceSearchStrategy(
                new BigDecimal(1), new BigDecimal(5));

        // when
        Order order1 = getOrderWithCertainPrice(new BigDecimal(4));
        Order order2 = getOrderWithCertainPrice(new BigDecimal(6));

        //then
        assertTrue(orderPriceSearchStrategy.filter(order1));
```

```
        assertFalse(orderPriceSearchStrategy.filter(order2));
    }
}
```

**Klasa ProductNameSearchStrategyTest**

```
public class ProductNameSearchStrategyTest {
    private Order getOrderWithCertainProductName(String name){
        Product product = mock(Product.class);
        given(product.getName()).willReturn(name);
        List<Product> products = new ArrayList<>();
        products.add(product);
        Order order = mock(Order.class);
        given(order.getProducts()).willReturn(products);
        return order;
    }

    @Test
    void testSearch(){
        //given
        ProductNameSearchStrategy productNameSearchStrategy = new
ProductNameSearchStrategy("Marchewka");

        // when
        Order order1 = getOrderWithCertainProductName("Marchewka");
        Order order2 = getOrderWithCertainProductName("Ogorek");

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