Projektowanie obiektowe

Laboratorium 5

Testy jednostkowe

Albert Gierlach

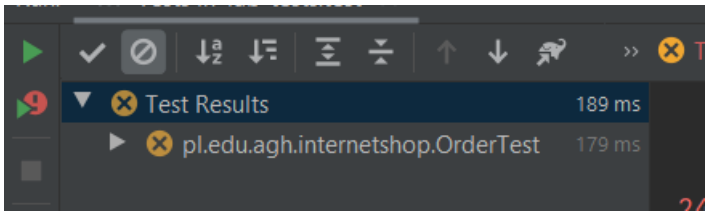
1. Zmienić wartość procentowa naliczanego podatku z 22% na 23%. Należy zweryfikować przypadki brzegowe przy zaokrągleniach.

Na początku zmodyfikowałem test tak, aby odpowiadał sytuacji gdy podatek wyniki 23%.

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

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

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

Pozostałe testy pozostały bez zmian.

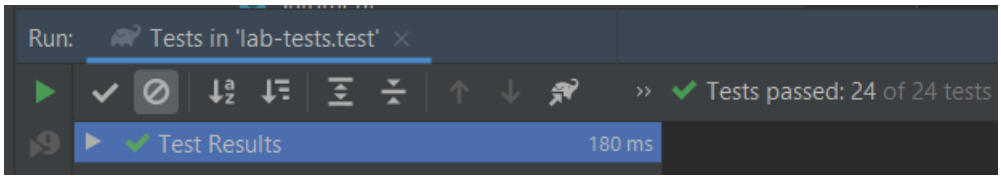
Po uruchomieniu oczywiście zmodyfikowany test zakańcza się niepowodzeniem.



Implementacja zmienionej wartości podatku polegała na zmianie stałej w klasie Order:

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

Po uruchomieniu wszystkie testy zakończyły się sukcesem.



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

Zmieniłem implementację klasy Order, tak aby przystosować jej interfejs do implementacji funkcjonalności.

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

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

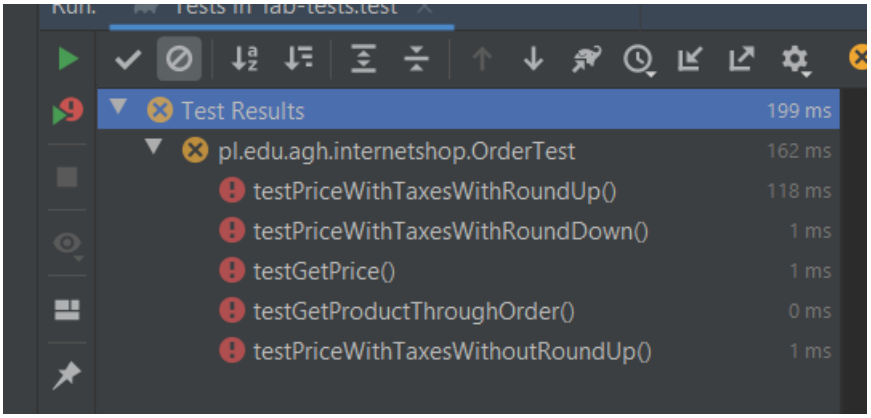
```
    public BigDecimal getPrice() {
//          return product.getPrice();
        return null;
    }

    public BigDecimal getPriceWithTaxes()
        return getPrice().multiply(TAX_VAL
    }

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

Po modyfikacji implementacji dostosowałem testy do bieżącego interfejsu klasy Order.

```java
public class OrderTest {

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

    @Test
    public void testGetProductThroughOrder() {
        // given
        Product expectedProduct = mock(Product.class);
        Product expectedProduct2 = mock(Product.class);
        Order order = new Order(Arrays.asList(expectedProduct, expectedProduct2));

        // when
        Product actualProduct = order.getProducts().get(0);
        Product actualProduct2 = order.getProducts().get(1);

        // then
        assertSame(expectedProduct, actualProduct);
        assertSame(expectedProduct2, actualProduct2);
    }

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

```java
@Test
public void testGetPrice() throws Exception {
    // given
    BigDecimal expectedTotalPrice = BigDecimal.valueOf(1666);
    BigDecimal expectedProductPrice = BigDecimal.valueOf(1000);
    BigDecimal expectedProductPrice2 = BigDecimal.valueOf(666);
    Product product = mock(Product.class);
    Product product2 = mock(Product.class);
    given(product.getPrice()).willReturn(expectedProductPrice);
    given(product2.getPrice()).willReturn(expectedProductPrice2);
    Order order = new Order(Arrays.asList(product, product2));

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

    // then
    assertBigDecimalCompareValue(expectedTotalPrice, actualTotalPrice);
}

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));
}
```

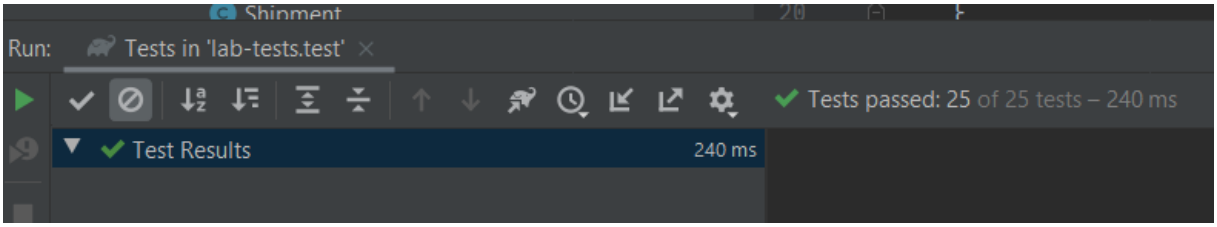
Uruchomienie testów skutkuje porażką:



Następnie przystąpiłem do implementacji właściwej funkcjonalności klasy Order.

```java
public BigDecimal getPrice() {
    return productList
            .stream()
            .map(Product::getPrice)
            .reduce(BigDecimal.valueOf(0), BigDecimal::add);
}

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

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

Po uruchomieniu testów:



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

Stworzone testy:

```java
@Test
public void testGetProductPriceWithDiscount(){
    // given
    BigDecimal expectedTotalPrice = BigDecimal.valueOf(900);
    BigDecimal expectedProductPrice = BigDecimal.valueOf(1000);
    BigDecimal discountTotal = BigDecimal.valueOf(0.1);
    Product product = new Product( name: "asdf", expectedProductPrice, discountTotal);

    //when
    BigDecimal actualTotalPrice = product.getPrice();

    //then
    assertBigDecimalCompareValue(expectedTotalPrice, actualTotalPrice);
}

@Test
public void testGetTotalPriceWithDiscount() throws Exception {
    // given
    BigDecimal expectedTotalPrice = BigDecimal.valueOf(1499.4);
    BigDecimal expectedProductPrice = BigDecimal.valueOf(1000);
    BigDecimal expectedProductPrice2 = BigDecimal.valueOf(666);
    BigDecimal discountTotal = BigDecimal.valueOf(0.1);
    Product product = mock(Product.class);
    Product product2 = mock(Product.class);
    given(product.getPrice()).willReturn(expectedProductPrice);
    given(product2.getPrice()).willReturn(expectedProductPrice2);
    Order order = new Order(Arrays.asList(product, product2), discountTotal);

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

    // then
    assertBigDecimalCompareValue(expectedTotalPrice, actualTotalPrice);
}

@Test
public void testInvalidDiscount() throws IllegalArgumentException{
    // when
    // given
    assertThrows(IllegalArgumentException.class,
            () -> new Product( name: "asdf", BigDecimal.valueOf(1), BigDecimal.valueOf(-1)));

    assertThrows(IllegalArgumentException.class,
            () -> new Product( name: "asdf2", BigDecimal.valueOf(1), BigDecimal.valueOf(2133)));
}
```

Następnie zaimplementowałem wymagane funkcjonalności:

```java
public class Product {

    public static final int PRICE_PRECISION = 2;
    public static final int ROUND_STRATEGY = BigDecimal.ROUND_HALF_UP;
    private BigDecimal discount = BigDecimal.ZERO;
    private final String name;
    private final BigDecimal price;

    public Product(String name, BigDecimal price) {...}

    public Product(String name, BigDecimal price, BigDecimal discount) {
        this(name, price);
        setDiscount(discount);
    }

    public String getName() {...}

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

    public BigDecimal getDiscount(){
        return discount;
    }

    public void setDiscount(BigDecimal d) throws IllegalArgumentException{
        if(d.compareTo(BigDecimal.ZERO) < 0 || d.compareTo(BigDecimal.ONE) > 0){
            throw new IllegalArgumentException("Invalid discount value");
        }

        this.discount = d;
        this.discount.setScale(PRICE_PRECISION, ROUND_STRATEGY);
    }
```

```java
public class Order {
    public static final int PRICE_PRECISION = 2;
    public static final int ROUND_STRATEGY = BigDecimal.ROUND_HALF_UP;
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
    private final UUID id;
    private final List<Product> productList;
    private boolean paid;
    private Shipment shipment;
    private ShipmentMethod shipmentMethod;
    private PaymentMethod paymentMethod;
    private BigDecimal discount = BigDecimal.ZERO;

    public Order(List<Product> product, BigDecimal discount) {
        this(product);
        setDiscount(discount);
    }

    public Order(List<Product> product) {...}

    public BigDecimal getPrice() {
        return productList
                .stream() Stream<Product>
                .map(Product::getPrice) Stream<BigDecimal>
                .reduce(BigDecimal.valueOf(0), BigDecimal::add) BigDecimal
                .multiply(BigDecimal.ONE.subtract(discount));
    }
```

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

  public void setDiscount(BigDecimal d) throws IllegalArgumentException{
      if(d.compareTo(BigDecimal.ZERO) < 0 || d.compareTo(BigDecimal.ONE) > 0){
          throw new IllegalArgumentException("Invalid discount value");
      }

      this.discount = d;
      this.discount.setScale(PRICE_PRECISION, ROUND_STRATEGY);
  }
```

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

Wykorzystując kod z poprzedniego laboratorium utworzyłem klasy odpowiadające za poszczególne kryteria wyszukiwania oraz klasę kompozytową, która umożliwi obsługę wielu kryteriów. Dodałem także klasę przechowującą historię zamówień (zaimplementowany tylko interfejs klasy). Następnie zaimplementowałem testy walidujące poprawności implementacji.

```
public class CustomerNameSearchStrategyTest {
    private Order getOrderWithCustomerName(String name) {
        Address a = mock(Address.class);
        Shipment s = mock(Shipment.class);
        Order o = mock(Order.class);

        given(a.getName()).willReturn(name);
        given(s.getRecipientAddress()).willReturn(a);
        given(o.getShipment()).willReturn(s);

        return o;
    }

    @Test
    void testFilter() {
        // given
        CustomerNameSearchStrategy orderPriceSearchStrategy = new CustomerNameSearchStrategy("Franek");

        // when
        Order order1 = getOrderWithCustomerName("Franek");
        Order order2 = getOrderWithCustomerName("Nie-Franek");

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

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

    @Test
    void testFilter() {
        // given
        BigDecimal min = BigDecimal.valueOf(2);
        BigDecimal max = BigDecimal.valueOf(10);
        PriceSearchStrategy orderPriceSearchStrategy = new PriceSearchStrategy(min, max);

        // when
        Order order1 = getOrderWithPrice(BigDecimal.valueOf(3));
        Order order2 = getOrderWithPrice(BigDecimal.valueOf(11));
        Order order3 = getOrderWithPrice(BigDecimal.valueOf(1));
        Order order4 = getOrderWithPrice(BigDecimal.valueOf(-1));
        Order order5 = getOrderWithPrice(BigDecimal.valueOf(0));
        Order order6 = getOrderWithPrice(min);
        Order order7 = getOrderWithPrice(max);

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

```java
public class ProductNameSearchStrategyTest {
    private Order getOrderContainsProductName(String name) {
        Product p = mock(Product.class);
        given(p.getName()).willReturn(name);
        return new Order(Collections.singletonList(p));
    }

    @Test
    void testFilter() {
        // given
        ProductNameSearchStrategy orderPriceSearchStrategy = new ProductNameSearchStrategy("Mleko");

        // when
        Order order1 = getOrderContainsProductName("Mleko");
        Order order2 = getOrderContainsProductName("Nie-Mleko");

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

```java
public class CompositeSearchStrategyTest {
    private Order getOrderWithParams(String customerName, String productName, BigDecimal price){
        Order order = mock(Order.class);
        Product p = mock(Product.class);
        Address a = mock(Address.class);
        Shipment s = mock(Shipment.class);

        given(a.getName()).willReturn(customerName);
        given(s.getRecipientAddress()).willReturn(a);

        given(p.getName()).willReturn(productName);

        given(order.getShipment()).willReturn(s);
        given(order.getPriceWithTaxes()).willReturn(price);
        given(order.getProducts()).willReturn(Collections.singletonList(p));
        return order;

    }

    @Test
    public void testFiltering(){
        // given
        BigDecimal min = BigDecimal.valueOf(2);
        BigDecimal max = BigDecimal.valueOf(10);
        CustomerNameSearchStrategy searchStrategy1 = new CustomerNameSearchStrategy("Franek");
        ProductNameSearchStrategy searchStrategy2 = new ProductNameSearchStrategy("Mleko");
        PriceSearchStrategy searchStrategy3 = new PriceSearchStrategy(min, max);
        CompositeSearchStrategy compositeSearchStrategy = new CompositeSearchStrategy();
        compositeSearchStrategy.addStrategy(searchStrategy1);
        compositeSearchStrategy.addStrategy(searchStrategy2);
        compositeSearchStrategy.addStrategy(searchStrategy3);

        // when
        Order order1 = getOrderWithParams( customerName: "Franek",  productName: "Mleko", BigDecimal.valueOf(20));
        Order order2 = getOrderWithParams( customerName: "Nie-Franek",  productName: "Ziemniaki", BigDecimal.valueOf(2));
        Order order3 = getOrderWithParams( customerName: "Franek",  productName: "Muszyna", BigDecimal.valueOf(4));
        Order order4 = getOrderWithParams( customerName: "Staszek",  productName: "Chleb", BigDecimal.valueOf(11));
        Order order5 = getOrderWithParams( customerName: "Franek",  productName: "Mleko", BigDecimal.valueOf(5));

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

```java
public class OrderHistoryTest {
    private Order getOrderWithParams(String customerName, String productName, BigDecimal price){
        Order order = mock(Order.class);
        Product p = mock(Product.class);
        Address a = mock(Address.class);
        Shipment s = mock(Shipment.class);

        given(a.getName()).willReturn(customerName);
        given(s.getRecipientAddress()).willReturn(a);

        given(p.getName()).willReturn(productName);

        given(order.getShipment()).willReturn(s);
        given(order.getPriceWithTaxes()).willReturn(price);
        given(order.getProducts()).willReturn(Collections.singletonList(p));
        return order;
    }
    @Test
    public void testHistoryList(){
        // given
        Order o1 = mock(Order.class);
        Order o2 = mock(Order.class);
        Order o3 = mock(Order.class);
        List<Order> expectedOrders = Arrays.asList(o1, o2, o3);
        OrderHistory orderHistory = new OrderHistory();

        // when
        orderHistory.addOrder(o1);
        orderHistory.addOrder(o2);
        orderHistory.addOrder(o3);

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

```java
@Test
public void testSearch() {
    BigDecimal min = BigDecimal.valueOf(2);
    BigDecimal max = BigDecimal.valueOf(10);
    CustomerNameSearchStrategy searchStrategy1 = new CustomerNameSearchStrategy("Franek");
    ProductNameSearchStrategy searchStrategy2 = new ProductNameSearchStrategy("Mleko");
    PriceSearchStrategy searchStrategy3 = new PriceSearchStrategy(min, max);
    CompositeSearchStrategy compositeSearchStrategy = new CompositeSearchStrategy();
    compositeSearchStrategy.addStrategy(searchStrategy1);
    compositeSearchStrategy.addStrategy(searchStrategy2);
    compositeSearchStrategy.addStrategy(searchStrategy3);
    Order order1 = getOrderWithParams( customerName: "Franek",  productName: "Mleko", BigDecimal.valueOf(20));
    Order order2 = getOrderWithParams( customerName: "Nie-Franek",  productName: "Ziemniaki", BigDecimal.valueOf(2));
    Order order3 = getOrderWithParams( customerName: "Franek",  productName: "Muszyna", BigDecimal.valueOf(4));
    Order order4 = getOrderWithParams( customerName: "Staszek",  productName: "Chleb", BigDecimal.valueOf(11));
    Order order5 = getOrderWithParams( customerName: "Franek",  productName: "Mleko", BigDecimal.valueOf(5));
    List<Order> expectedOrders = Collections.singletonList(order5);
    OrderHistory orderHistory = new OrderHistory();

    // when
    orderHistory.addOrder(order1);
    orderHistory.addOrder(order2);
    orderHistory.addOrder(order3);
    orderHistory.addOrder(order4);
    orderHistory.addOrder(order5);

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

Po zaimplementowaniu wszystkich testów przystąpiłem do implementacji metod w stworzonych klasach. Struktury klas prezentują się następująco:

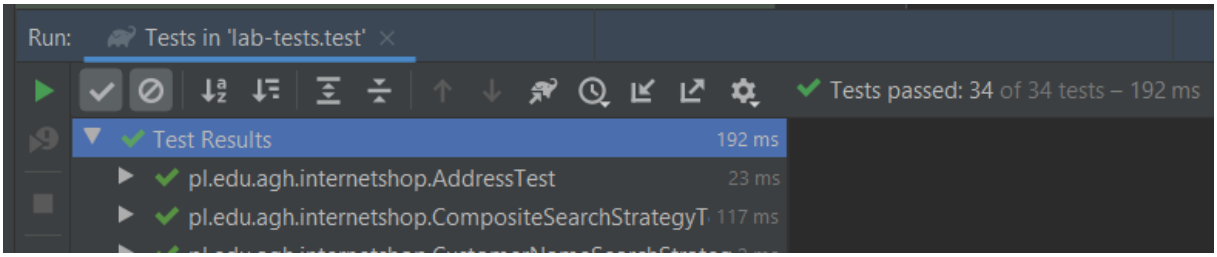
```java
package pl.edu.agh.internetshop;

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

```java
public class OrderHistory {
    private final List<Order> orderList = new LinkedList<>();

    public void addOrder(Order o){
        this.orderList.add(o);
    }

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

    public List<Order> search(CompositeSearchStrategy searchStrategy){
        return orderList.stream()
                .filter(searchStrategy::filter)
                .collect(Collectors.toList());
    }
}
```

```java
public class CompositeSearchStrategy implements SearchStrategy {
    List<SearchStrategy> searchStrategies;

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

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

    @Override
    public boolean filter(Order o) {
        return searchStrategies
                .stream()
                .allMatch(searchStrategy -> searchStrategy.filter(o));
    }
}
```

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

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

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

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

    public PriceSearchStrategy(BigDecimal min, BigDecimal max){
        this.max = max;
        this.min = min;
    }

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



Jak widać testy przebiegły pomyślnie.

Wnioski:

Technika TDD pozwala na wcześniejsze wykrywanie błędów oraz zaoszczędzenie cennego czasu developerów oraz testerów, gdyż błąd zostanie poprawiony od razu przy implementacji testowanego kody. Należy pamiętać jednak, że TDD nie gwarantuje całkowitego uniknięcia błędów, ale pozwala na zminimalizowanie zakresu ich występowania. Wadą takiego rozwiązania jest niewątpliwie czasochłonność, jednak w porównaniu z korzyściami jakie przynosi jest warta stosowania.