Projektowanie Obiektowe

Laboratorium 5

Konrad Siuzdak, Paweł Kocimski

Zadanie 1

Zmieniono test na:

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

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

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

oraz podatek:

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

Zadanie 2

Najpierw zmieniono klasę Order - pojedynczy produkt na listę produktów.

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

oraz wszystkie metody w klasie, które wskazał Intellij jako błędne ze względu na zmieniony typ.

Następnie zmodyfikowano testy tak, aby wykonywały się dla list produktów, które są jednoelementowe:

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

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

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

  // then
  assertSame(expectedProduct, actualProductList);
}
@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));
}
```

Zadanie 3

Zmodyfikowano klasę 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.ZERO;

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

   public BigDecimal getDiscount() {
      return discount;
   }

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

   public String getName() {
      return name;
   }

   public BigDecimal getPrice() {
      return price;
   }

   public BigDecimal getDiscountedPrice()
   {
      return price;
   }
}
```

Dodano testy:

```
@Test
public void testDiscount()
{
  //given
  Product product=new Product("anyName",BigDecimal.TEN);
  final BigDecimal discount= BigDecimal.valueOf(0.2);
  product.setDiscount(discount);

  //when
  final BigDecimal actualDiscount=product.getDiscount();

  //then
  assertBigDecimalCompareValue(actualDiscount,discount);
}

@Test
public void testGetDiscountedPrice()
{
  //given
  Product p=new Product("anyName",BigDecimal.TEN);
  final BigDecimal actualDiscount= BigDecimal.valueOf(0.2);
  p.setDiscount(actualDiscount);

  //when
  final BigDecimal discountedPrice=p.getDiscountedPrice();
  System.out.println(discountedPrice);

  //then
  assertBigDecimalCompareValue(BigDecimal.valueOf(8.0),discountedPrice);
}
```

Drugi test nie przeszedł, więc poprawiono implementację:

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

Dla klasy Order:

```
public BigDecimal getDiscountedPrice()
{
   return this.getPrice().subtract(this.getPrice().multiply(discount));
}

@Test
public void testGetDiscountedPrice()
{
  //given
  Product p1=new Product("anyName",BigDecimal.TEN);
  Product p2=new Product("anyName",BigDecimal.TEN);
  List<Product> products=new ArrayList<>();
  products.add(p1);
  products.add(p2);
  Order order=new Order(products);
  order.setDiscount(BigDecimal.valueOf(0.2));

  //when
  final BigDecimal priceAfterDiscount=order.getDiscountedPrice();

  //then
  assertBigDecimalCompareValue(BigDecimal.valueOf(16),priceAfterDiscount);
}
```

Zadanie 4

Dodano do klasy Order pole Surname, aby móc przypisać zamówienie do danego nazwiska i wyszukiwać po nim. Do pola dodano getter.

```
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 String surname;

   public Order(List<Product> products) {
      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() {
     return products.stream().map(product ->
product.getPrice()).reduce(BigDecimal.ZERO,BigDecimal::add);
  }

  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 String getSurname(){
     return this.surname;
```

```
  }
}
```

Następnie utworzono interfejs Search umożliwiający wyszukiwanie

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

Dodano implementujące klasy, które umożliwiają wyszukiwanie po czterech parametrach (nazwa produktu, cena zamówienia, nazwisko zamawiającego

```
public class SearchProductName implements Search {
   private String productName;

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

   @Override
   public boolean filter(Order order) {
      List<Product> products = order.getProducts();
      for (Product product: products) {
         if (product.getName().equals(this.productName)) {
            return true;
         }
      }
      return false;
   }
}
```

```
public class SearchOrderPrice implements Search{
   BigDecimal orderPrice;


   public SearchOrderPrice(BigDecimal price) {
      this.orderPrice = price;
   }

   @Override
   public boolean filter(Order order) {
      return order.getPriceWithTaxes().compareTo(this.orderPrice) == 0;
   }
}
```

```
public class SearchSurnameOrderer implements Search{
  private String surnameOrderer;


  public SearchSurnameOrderer(String surnameOrderer) {
    this.surnameOrderer = surnameOrderer;
  }

  @Override
  public boolean filter(Order order) {
    if (order.getSurname() != null) {
      return order.getSurname().equals(this.surnameOrderer);
    }
    return false;
  }
}
```

Dodano klasę umożliwiającą wyszukiwanie po wielu parametrach. Zadanie zrealizowano przy użyciu wzorca kompozyt.

```
public class CompositeSearch implements Search{
  private final List<Search> filters;

  public CompositeSearch(List<Search> filters) {
    this.filters = filters;
  }

  @Override
  public boolean filter(Order order) {
    boolean result = true;
    for(Search filter: filters){
      result = result && filter.filter(order);
    }
    return result;
  }
}
```

Do każdej klasy napisano testy

```
public class SearchProductNameTest {
  private Order getMockedOrder() {
    Order order = mock(Order.class);
    List<Product> productList = Arrays.asList(
        new Product("milk", BigDecimal.valueOf(2.89)),
        new Product("cheese", BigDecimal.valueOf(23.45))
    );
    given(order.getProducts()).willReturn(productList);
    return order;
  }
```

```
  @Test
  public void testExistsInList() {
     // given
     Order order = getMockedOrder();

     // when
     SearchProductName search = new SearchProductName("milk");

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

  @Test
  public void testNotExistsInList() {
     // given
     Order order = getMockedOrder();

     // when
     SearchProductName search = new SearchProductName("cheese");

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

```
public class SearchOrderPriceTest {
  private Order getMockedOrder() {
     Order order = mock(Order.class);
     given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(15));
     return order;
  }

  @Test
  public void testSameOrderPrice() {
     // given
     Order order = getMockedOrder();

     // when
     SearchOrderPrice search = new SearchOrderPrice(BigDecimal.valueOf(15));

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

  @Test
  public void testNotSameOrderPrice() {
     // given
     Order order = getMockedOrder();

     // when
     SearchOrderPrice search = new SearchOrderPrice(BigDecimal.valueOf(7));
```

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

```
public class SearchSurnameOrdererTest {
    @Test
    public void testSameSurnameOrderer() {
        // given
        Order order = mock(Order.class);
        given(order.getSurname()).willReturn("Kowalski");

        // when
        SearchSurnameOrderer search = new SearchSurnameOrderer("Kowalski");

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

    @Test
    public void testNotSameSurnameOrderer() {
        // given
        Order order = mock(Order.class);
        given(order.getSurname()).willReturn("Nowak");

        // when
        SearchSurnameOrderer search = new SearchSurnameOrderer("Kowalski");

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

    @Test
    public void testSurnameOrdererIsNull() {
        // given
        Order order = mock(Order.class);
        given(order.getSurname()).willReturn(null);

        // when
        SearchSurnameOrderer search = new SearchSurnameOrderer("Kowalski");

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

```
public class CompositeSearchTest {
    private Order getMockedOrder() {
        Order order = mock(Order.class);
        List<Product> mockedProducts = Arrays.asList(
                new Product("Eggs", BigDecimal.valueOf(0.59)),
```

```
            new Product("meat", BigDecimal.valueOf(24.43))
    );
    given(order.getProducts()).willReturn(mockedProducts);
    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(14));
    given(order.getSurname()).willReturn("Kowalski");
    return order;
}
@Test
public void sameAllParametersNotProductName() {
    // given
    Order order = getMockedOrder();
    Search searchProductName = new SearchProductName("milk");
    Search searchSurnameOrderer = new SearchSurnameOrderer("Kowalski");
    Search searchOrderPrice = new SearchOrderPrice(BigDecimal.valueOf(14));

    // when
    CompositeSearch search = new CompositeSearch(
            Arrays.asList(searchProductName, searchSurnameOrderer, searchOrderPrice));

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

@Test
public void sameAllParametersNotOrdererSurname() {
    // given
    Order order = getMockedOrder();
    Search searchProductName = new SearchProductName("eggs");
    Search searchSurnameOrderer = new SearchSurnameOrderer("Nowak");
    Search searchOrderPrice = new SearchOrderPrice(BigDecimal.valueOf(14));

    // when
    CompositeSearch search = new CompositeSearch(
            Arrays.asList(searchProductName, searchSurnameOrderer, searchOrderPrice));

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

@Test
public void sameAllParametersNotTotalPrice() {
    // given
    Order order = getMockedOrder();
    Search searchProductName = new SearchProductName("eggs");
    Search searchSurnameOrderer = new SearchSurnameOrderer("Nowak");
    Search searchOrderPrice = new SearchOrderPrice(BigDecimal.valueOf(13));

    // when
    CompositeSearch search = new CompositeSearch(
            Arrays.asList(searchProductName, searchSurnameOrderer, searchOrderPrice));

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

@Test
public void sameAllParameters() {
```

```
        // given
        Order order = getMockedOrder();
        Search SearchProductName = new SearchProductName("Eggs");
        Search SearchSurnameOrderer = new SearchSurnameOrderer("Kowalski");
        Search SearchOrderPrice = new SearchOrderPrice(BigDecimal.valueOf(14));

        // when
        CompositeSearch search = new CompositeSearch(
                Arrays.asList(SearchProductName, SearchSurnameOrderer, SearchOrderPrice));

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

Następnie dodano klasę OrdersHistory przechowującą w swoim polu zamówienia z przeszłości. Zaimplementowano ją tak, aby móc wyszukać zamówienia spełniające określone kryteria.

```
public class OrdersHistory {
    private final List<Order> pastOrders;

    public OrdersHistory(List<Order> pastOrders) {
        this.pastOrders = Objects.requireNonNull(pastOrders, "pastOrders cannot be null");
        this.pastOrders.forEach((p) -> Objects.requireNonNull(p,"order cannot be null"));
    }

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

    public List<Order> getPastOrders() {
        return pastOrders;
    }

    public List<Order> getFilteredOrders(Search search) {
        List<Order> filteredList = new ArrayList<>();
        for (Order order: pastOrders) {
            if (search.filter(order)) {
                filteredList.add(order);
            }
        }
        return filteredList;
    }
}
```

Do powyższej klasy napisano testy

```
public class OrdersHistoryTest {

    @Test
    public void testPastPastOrdersListIsNull() {
```

```
    // when then
    assertThrows(NullPointerException.class, () -> new OrdersHistory(null));
}

@Test
public void testPastOrdersListIsNull() {
    // given
    List<Order> pastOrders = Arrays.asList(mock(Order.class), null);

    // when then
    assertThrows(NullPointerException.class, () -> new OrdersHistory(pastOrders));
}

@Test
void testGetMultipleOrders() {
    // given
    List<Order> orders = Arrays.asList(mock(Order.class), mock(Order.class));

    // when
    OrdersHistory ordersHistory = new OrdersHistory(orders);

    // then
    assertEquals(2, ordersHistory.getPastOrders().size());
    assertSame(orders.get(0), ordersHistory.getPastOrders().get(0));
    assertSame(orders.get(1), ordersHistory.getPastOrders().get(1));
}

@Test
void testGetPastOrdersWithAddingOrders() {
    // given
    Order expectedOrder = mock(Order.class);

    // when
    OrdersHistory ordersHistory = new OrdersHistory(new ArrayList<>());
    ordersHistory.addOrder(expectedOrder);

    // then
    assertEquals(1, ordersHistory.getPastOrders().size());
    assertSame(expectedOrder, ordersHistory.getPastOrders().get(0));
}




@Test
void testGetFilteredOrdersWithProductName() {
    // given
    Product product = mock(Product.class);
    Product product2 = mock(Product.class);
    Product product3 = mock(Product.class);
    Product product4 = mock(Product.class);

    Order order = mock(Order.class);
    Order order1 = mock(Order.class);
    Order order2 = mock(Order.class);
```

```
    given(product.getName()).willReturn("cheese");
    given(product2.getName()).willReturn("eggs");
    given(product3.getName()).willReturn("milk");
    given(product4.getName()).willReturn("meat");

    given(order.getProducts()).willReturn(Arrays.asList(product, product2));
    given(order1.getProducts()).willReturn(Arrays.asList(product2, product3));
    given(order2.getProducts()).willReturn(Arrays.asList(product, product2, product2, product3));

    Search search = new SearchProductName("cheese");

    // when
    OrdersHistory ordersHistory = new OrdersHistory(Arrays.asList(order, order1, order2));

    // then
    assertEquals(2, ordersHistory.getFilteredOrders(search).size());
    assertSame(order, ordersHistory.getFilteredOrders(search).get(0));
    assertSame(order2, ordersHistory.getFilteredOrders(search).get(1));
}

@Test
void getFilteredOrdersWithOrdererSurname() {
    // given
    Order order = mock(Order.class);
    Order order2 = mock(Order.class);
    Order order3 = mock(Order.class);

    given(order.getSurname()).willReturn("Kowalski");
    given(order2.getSurname()).willReturn("Nowak");
    given(order3.getSurname()).willReturn("Kowalski");

    Search search = new SearchSurnameOrderer("Kowalski");

    // when
    OrdersHistory ordersHistory = new OrdersHistory(Arrays.asList(order, order2, order3));

    // then
    assertEquals(2, ordersHistory.getFilteredOrders(search).size());
    assertSame(order, ordersHistory.getFilteredOrders(search).get(0));
    assertSame(order3, ordersHistory.getFilteredOrders(search).get(1));
}

@Test
void getFilteredOrdersWithOrderPrice() {
    // given
    Order order = mock(Order.class);
    Order order2 = mock(Order.class);
    Order order3 = mock(Order.class);

    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(3));
    given(order2.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(5));
    given(order3.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(3));

    Search search = new SearchOrderPrice(BigDecimal.valueOf(5));

    // when
    OrdersHistory ordersHistory = new OrdersHistory(Arrays.asList(order, order2, order3));
```

```
        // then
        assertEquals(1, ordersHistory.getFilteredOrders(search).size());
        assertSame(order2, ordersHistory.getFilteredOrders(search).get(0));
    }

    @Test
    public void testSearchIsNull() {
        // given

        // when
        OrdersHistory ordersHistory = new OrdersHistory(Arrays.asList(mock(Order.class),
mock(Order.class)));

        // then
        assertThrows(NullPointerException.class, () -> ordersHistory.getFilteredOrders(null));
    }

    @Test
    void getCompositeFilteredOrders() {
        // given
        Product product = mock(Product.class);
        Product product2 = mock(Product.class);
        Product product3 = mock(Product.class);
        Product product4 = mock(Product.class);

        Order order = mock(Order.class);
        Order order1 = mock(Order.class);
        Order order2 = mock(Order.class);

        given(product.getName()).willReturn("cheese");
        given(product2.getName()).willReturn("eggs");
        given(product3.getName()).willReturn("milk");
        given(product4.getName()).willReturn("meat");

        given(order.getProducts()).willReturn(Arrays.asList(product, product2, product3));
        given(order1.getProducts()).willReturn(Arrays.asList(product2, product3));
        given(order2.getProducts()).willReturn(Arrays.asList(product, product2, product2, product3));

        given(order.getSurname()).willReturn("Kowalski");
        given(order1.getSurname()).willReturn("Nowak");
        given(order2.getSurname()).willReturn("Kowalski");

        given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(3));
        given(order1.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(5));
        given(order2.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(5));

        Search search = new CompositeSearch(Arrays.asList(
                new SearchProductName("eggs"),
                new SearchSurnameOrderer("Kowalski"),
                new SearchOrderPrice(BigDecimal.valueOf(5))
        ));

        // when
        OrdersHistory ordersHistory = new OrdersHistory(Arrays.asList(order, order1, order2));

        // then
```

```
        assertEquals(1, ordersHistory.getFilteredOrders(search).size());
        assertSame(order2, ordersHistory.getFilteredOrders(search).get(0));
    }
}
```