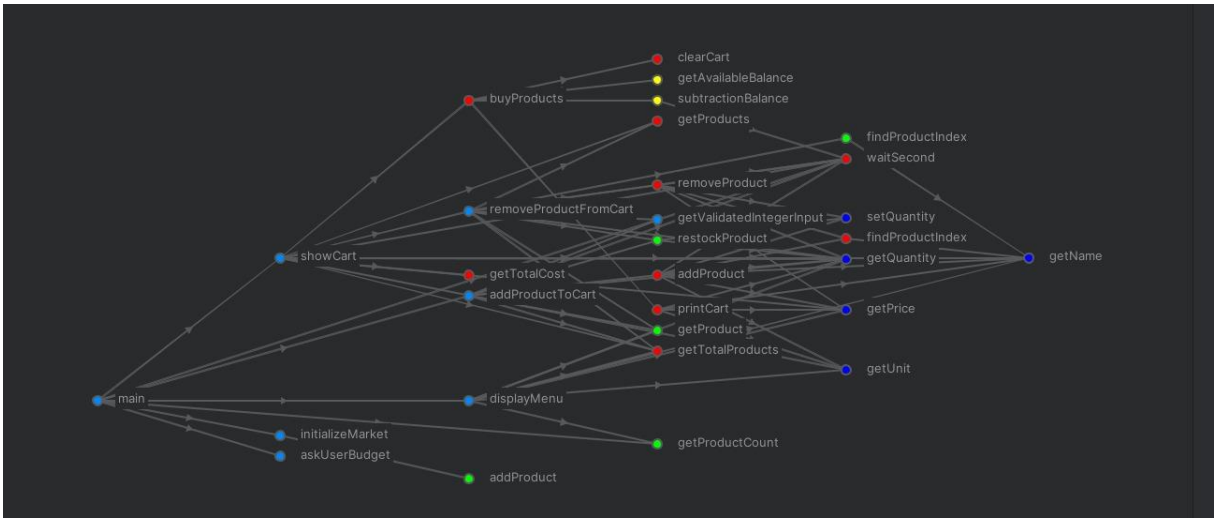
SENG 453

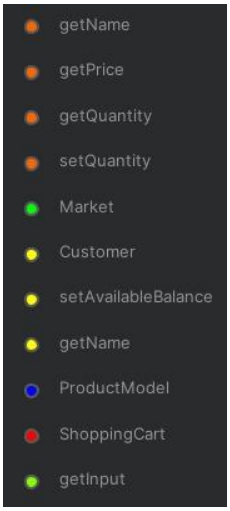
HW3 REPORT

Alperen Göyce (39028411222)

Mehlika Eroğlu (58717180232)

1.Call Graph





2.Code and Executions

- The ShoppingCart class manages the products selected by the customer. It interacts directly with the Market for stock updates and the Customer for purchase transactions.

testAddProductToCart verifies that products can be added to the cart and the market stock is appropriately reduced.

Relationship Tested: Interaction between ShoppingCart (to add products) and Market (to update stock).

```
@Test
public void testAddProductToCart() {

    Market mockMarket = Mockito.mock(Market.class);
    ShoppingCart mockCart = Mockito.mock(ShoppingCart.class);

    ProductModel mockProduct = new ProductModel("Apple", 10, 10, "kg");

    Mockito.when(mockMarket.getProduct(0)).thenReturn(mockProduct);
    Mockito.when(mockCart.getTotalProducts()).thenReturn(0);

    ProductModel product = mockMarket.getProduct(0);
    mockCart.addProduct(product, 5);
    mockMarket.restockProduct(0, -5);

    Mockito.verify(mockCart, Mockito.times(1)).addProduct(mockProduct, 5);
    Mockito.verify(mockMarket, Mockito.times(1)).restockProduct(0, -5);

    System.out.println("Product added to cart and market stock updated.");
}
```

testRemoveProductFromCart tests the removal of products from the cart and restocking in the market.

Relationship Tested: Interaction between ShoppingCart (to remove products) and Market (to restock products).

```
@Test
void testRemoveProductFromCart() {

    ProductModel mockProduct = new ProductModel("Banana", 3, 5, "kg");
    when(mockCart.getProducts()).thenReturn(new ProductModel[]{mockProduct});
    when(mockCart.removeProduct(mockProduct, 2)).thenReturn(2);
    doNothing().when(mockMarket).restockProduct(0, 2);

    mockCart.removeProduct(mockProduct, 2);
    mockMarket.restockProduct(0, 2);

    verify(mockCart, times(1)).removeProduct(mockProduct, 2);
    verify(mockMarket, times(1)).restockProduct(0, 2);

    System.out.println("Product removed from cart and market stock updated.");
}
```

testBuyProducts validates that customers can successfully purchase products in the cart if they have sufficient balance.

Relationship Tested: Interaction between Customer (balance update) and ShoppingCart (finalizing purchases).

```
@Test
void testBuyProducts() {
    cart = new ShoppingCart();
    product1 = new ProductModel("Apple", 1, 1, "kg");
    product2 = new ProductModel("Banana", 2, 2, "kg");

    when(mockCustomer.getAvailableBalance()).thenReturn(10.0);

    mockCart.addProduct(product1, 1);
    mockCart.buyProducts(mockCustomer);
```

```
    verify(mockCustomer, times(1)).getAvailableBalance();
    verify(mockCustomer, times(1)).subtractionBalance(1);
    assertEquals(0, mockCart.getTotalProducts());
}
```

testClearCart ensures that the cart can be cleared, resetting its state.

```
@Test
void testClearCart() {
    mockCart.addProduct(product1, 1);
    mockCart.clearCart();

    assertEquals(0, mockCart.getTotalProducts());
    assertEquals(0, mockCart.getTotalCost());
}
```

- The Customer class interacts with the ShoppingCart during purchases and directly updates its available balance.

testCustomerBalanceUpdate ensures customer balance is reduced when making a purchase.

```
@Test
void testCustomerBalanceUpdate() {

    when(mockCustomer.getAvailableBalance()).thenReturn(50.0);
    doNothing().when(mockCustomer).subtractionBalance(20.0);

    mockCustomer.subtractionBalance(20.0);

    verify(mockCustomer, times(1)).subtractionBalance(20.0);

    System.out.println("Customer balance updated.");
}
```

testCheckout validates the checkout process, ensuring sufficient funds before purchase.

```
@Test
void testCheckout() {
    ProductModel mockProduct = new ProductModel("Milk", 10, 2, "liter");
    when(mockCart.getProducts()).thenReturn(new ProductModel[]{mockProduct});
    when(mockCart.getTotalCost()).thenReturn(20);
    when(mockCustomer.getAvailableBalance()).thenReturn(50.0);

    if (mockCustomer.getAvailableBalance() >= mockCart.getTotalCost()) {
        mockCart.buyProducts(mockCustomer);
    }
    verify(mockCart, times(1)).buyProducts(mockCustomer);
}
```

- The Market class is responsible for managing product stock and responding to queries about product availability.

testInvalidQuantityHandling verifies that invalid quantities (greater than available stock) are handled gracefully.

```
@Test
void testInvalidQuantityHandling() {

    ProductModel mockProduct = new ProductModel("Juice", 5, 0, "liter");
    when(mockMarket.getProduct(0)).thenReturn(mockProduct);

    int invalidQuantity = 10;
    if (mockProduct.getQuantity() < invalidQuantity) {
        System.out.println("Invalid quantity or product is out of stock.");
```

```
    }
    verify(mockCart, never()).addProduct(mockProduct, invalidQuantity);

    System.out.println("No product added to cart due to invalid quantity.");
}
```

testRestockProduct ensures that products in the market can be restocked.

Relationship Tested: Updates between Market and individual ProductModel.

```
@Test
void testRestockProduct() {
    market = new Market(10);

    when(product1.getName()).thenReturn("Apple");
    when(product1.getPrice()).thenReturn(5);
    when(product1.getQuantity()).thenReturn(10);
    when(product1.getUnit()).thenReturn("kg");

    mockMarket.addProduct(product1);
    mockMarket.restockProduct(0, 10);

    verify(product1, times(1)).setQuantity(20);
}
```

testFindProductIndex verifies that products can be located in the market's inventory.

```
@Test
void testFindProductIndex() {
    when(product1.getName()).thenReturn("Apple");
    when(product1.getPrice()).thenReturn(5);
    when(product1.getQuantity()).thenReturn(10);
    when(product1.getUnit()).thenReturn("kg");

    mockMarket.addProduct(product1);
    int index = mockMarket.findProductIndex(product1);

    assertEquals(0, index);
}
```

3.Relations Between Classes

ShoppingCart ↔ Market:

ShoppingCart ↔ Customer:

Market ↔ ProductModel:

4.Executions