SE 322 - SE 318
SOFTWARE VERIFICATION AND VALIDATION
SPRING 2023-2024

<E-COMMERCE-MS>

<BEDIRHAN ASAR, EMIR ERDEN, MIHAIL TALEV, SELCUK TALHA KUL,YASAR CAN>

UNIT TEST DOCUMENT

Version <*3.0*>

<31.05.2024>

VERSION HISTORY

VERSION 1.0 (date)
V1 Release Notes Following Requirements are implemented: The system should allow login to administrators. The system should allow login to customers. The system should allow administrators to add, edit and delete products from the online store. The system should allow administrators to manage customer accounts, including creating, editing, deleting and viewing order history. The system should allow customers to add items to their shopping carts. Full Changelog: https://github.com/serhatuzunbayir/ECommerce-MS/commits/V1
VERSION 2.0 (date)
V2 Release Notes Following Requirements are added to the system: The system should allow customers to filter shopping cards. The system should allow admin to filter shopping cards history.

19 test cases created using Junit, were added to the system. Full Changelog: V1...V2
VERSION 3.0 (date)
There is no new features in this versions. 22 test cases created using Junit, were added to the system. Test suite is added.

- **INTRODUCTION**
- **PURPOSE OF THE TEST CASE DOCUMENT**

The purpose of this Test Case Document is to outline and detail the functional requirements and testing procedures for the e-commerce management system. This document serves as a comprehensive guide for the project manager, project team, and testing team. It may also be shared with the client, users, and other stakeholders when their input or approval is necessary during the testing process.

- **CONSTRAINTS**

Java: The e-commerce management system is developed using Java. All test cases will be written in Java to maintain consistency with the project's codebase.
Build Automation: Maven will be used for build automation and dependency management. Maven's Surefire plugin will be configured to run the tests during the build process.
Spring Boot: The application is built using Spring Boot. Spring's testing support will be utilized to create context-aware tests, especially for integration and controller tests.

- **UNIT TEST FRAMEWORK: *<JUNIT>***

For the e-commerce management system project, we are using JUnit as the unit testing framework. JUnit is a widely-used framework in the Java ecosystem, known for its simplicity and robustness. It is essential for ensuring that our e-commerce platform operates reliably and efficiently. Key properties of JUnit relevant to our project include:

- Annotation-based Testing: JUnit uses annotations such as @Test, @Before, and @After to define test methods and setup/teardown procedures. This is crucial for organizing and managing our test cases effectively.
- Assertions: JUnit provides a rich set of assertions (e.g., assertEquals, assertTrue, assertNotNull) to verify expected outcomes. These assertions help validate the correctness of various e-commerce functionalities, such as user authentication, product management, and order processing.

- Test Runners: JUnit supports various test runners to execute tests, including support for running test suites. This allows us to group related tests and run them together, ensuring comprehensive coverage of the system's features.

- **TEST CASES**

Test Case 1: CategoryConstructorTest.java	
Test Definition	
testConstructorPositive: Validates the Category constructor's ability to correctly initialize fields with valid inputs. **testConstructorNegative: Validates the Category constructor's behavior when null inputs are provided, ensuring it sets fields to null appropriately.**	
Input Value	
setUp: **category: null** **testConstructorPositive:** **ID: null** **name: "Test Category" (String)** **description: "This is a test category description." (String)** **testConstructorNegative:** **ID: null** **name: null** **description: null**	
Expected Value	**Actual Value**
Null assertions validate null fields. Equals assertions validate equality.	**Result is as expected.**
Result of Test Case	***succesful***
Test Script	

// The testConstructorPositive method creates a Category instance with valid name and description,
// verifying that the fields are initialized correctly.
 	
// Create a category instance using the constructor
String name = "Test Category";
String description = "This is a test category description.";
category = new Category(name, description);
 	
// Verify that the fields are initialized correctly
assertNull("ID should be null", category.getId());
	// The testConstructorNegative method creates a Category instance with null name and description,
	// checking if the fields are initialized to null.
	
	// Create a category instance with null name and description
	category = new Category(null, null);
	
	// Verify that the fields are initialized correctly
	assertNull("ID should be null", category.getId());
	assertNull("Name should be null", category.getName());
	assertNull("Description should be null", category.getDescription());

assertEquals("Name should match", name, category.getName()); assertEquals("Description should match", description, category.getDescription());	

Test Case 2: RoleGetAuthorityTest.java	
Test Definition	
testGetAuthorityPositive: Verifies that the getAuthority() method of the Role enum returns the correct authority string for a valid role. **testGetAuthorityNegative: Validates that the getAuthority() method of the Role enum does not return an incorrect authority string for a valid role.**	
Input Value	
setUp: **role: null** **testGetAuthorityPositive:** **role: Role.ROLE_USER (enum)** **testGetAuthorityNegative:** **role: Role.ROLE_USER (enum)**	
Expected Value	**Actual Value**
Equals assertion validates equality. **False assertion validates inequality.**	**Result is as expected.**
Result of Test Case	***successful***
Test Script	

// Set up the role for positive test case
 role = Role.*ROLE_USER*;
 	
 // Call the method to get authority
 String authority = role.getAuthority();
 	
 // Assert that the authority matches the name of the enum
 assertEquals("ROLE_USER", authority);
	// Set up the role for negative test case
	 role = Role.*ROLE_USER*;
	
	 // Call the method to get authority
	 String authority = role.getAuthority();
	
	 // Assert that the authority does not match an incorrect value
	 assertFalse("ROLE_ADMIN".equals(authority));

Test Case 3: UserGetAuthoritiesTest.java
Test Definition
testGetAuthoritiesPositive: Validates that the getAuthorities() method of the User class returns a collection containing exactly one element, which corresponds to the user's

role. testGetAuthoritiesNegative: Verifies that the getAuthorities() method of the User class does not return an empty collection.	
Input Value	
setUp: username: "username" (String) password: "password" (String) name: "realName" (String) surName: "realSurname" (String) ua = (UserAddress) r = ROLE_USER (enum) testGetAuthorityPositive: role: Role.ROLE_USER (enum) testGetAuthorityNegative: role: Role.ROLE_USER (enum)	
Expected Value	**Actual Value**
Equals assertions validate equality. False assertion validates inequality.	Result is as expected.
Result of Test Case	*successful*
Test Script	

// Call the method to get authorities
Collection<? extends GrantedAuthority> authorities = u.getAuthorities();
 	
// Assert that the authorities list contains exactly one element which is the user's role
assertEquals(1, authorities.size());
assertEquals(Role.*ROLE_USER*, authorities.iterator().next())
	// Call the method to get authorities
	Collection<? extends GrantedAuthority> authorities = u.getAuthorities();
	
	// Assert that the authorities list is empty
	assertFalse(authorities.isEmpty());

Test Case 4: ManufacturerConstructorTest.java
Test Definition
testConstructorPositive: Validates that the Manufacturer constructor initializes the ID, name, and address fields correctly when valid inputs are provided. testConstructorNegative: Verifies that the Manufacturer constructor sets the ID, name, and address fields to null when null inputs are provided.
Input Value
setUp: manufacturer: null testConstructorPositive: ID: null name: "Test Manufacturer" (String)

address: "123 Test Address" (String) **<u>testConstructorNegative:</u>** **ID: null** **name: null** **address: null**	
Expected Value	**Actual Value**
Null assertions validate nullfields. Equals assertions validate equality.	**Result is as expected.**
Result of Test Case	***successful***
Test Script	

// The testConstructorPositive method creates a Manufacturer instance with valid name and address,
// verifying that the fields are initialized correctly.
 	
// Create a manufacturer instance using the constructor
String name = "Test Manufacturer";
String address = "123 Test Address";
manufacturer = new Manufacturer(name, address);
 	
// Verify that the fields are initialized correctly
assertNull("ID should be null", manufacturer.getId());
assertEquals("Name should match", name, manufacturer.getName());
assertEquals("Address should match", address, manufacturer.getAddress());
	// The testConstructorNegative method creates a Manufacturer instance with null name and address,
	// checking if the fields are initialized to null.
	
	// Create a manufacturer instance with null name and address
	manufacturer = new Manufacturer(null, null);
	
	// Verify that the fields are initialized correctly
	assertNull("ID should be null", manufacturer.getId());
	assertNull("Name should be null", manufacturer.getName());
	assertNull("Address should be null", manufacturer.getAddress());

Test Case 5: CategoryNotFoundExceptionTest.java	
Test Definition	
<u>testConstructorPositive</u>: Validates that the CategoryNotFoundException constructor correctly formats the exception message when initialized with a valid categoryId. **<u>testConstructorNegative</u>: Verifies that the CategoryNotFoundException constructor handles null categoryId gracefully by formatting the exception message appropriately.**	
Input Value	
<u>setUp:</u> **exception:null** **<u>testConstructorPositive:</u>** **categoryID: 123L(Long)** **<u>testConstructorNegative:</u>** **categoryID: null**	

Expected Value	Actual Value
Equals assertions validate equality.	**Result is as expected.**
Result of Test Case	***successful***
Test Script	

// The testConstructorPositive method creates a CategoryNotFoundException instance with a valid categoryId,
// verifying that the message is initialized correctly.
 	
// Define the category ID that will be used to construct the exception
Long categoryId = 123L;
 	
// Create the exception instance using the constructor
exception = new CategoryNotFoundException(categoryId);
 	
// Verify that the message is formatted correctly
String expectedMessage = String.*format*("Category with id %d does not exist.", categoryId);
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
	// The testConstructorNegative method creates a CategoryNotFoundException instance with a null categoryId,
	// checking if the message is handled correctly. We assume that the constructor should handle nulls gracefully.
	
	// Define a null category ID
	Long categoryId = null;
	
	// Create the exception instance using the constructor
	exception = new CategoryNotFoundException(categoryId);
	
	// Verify that the message is handled correctly
	String expectedMessage = "Category with id null does not exist.";
	assertEquals("Exception message should handle null ID", expectedMessage, exception.getMessage());

Test Case 6	
Test Definition	
testConstructorPositive: **Validates that the InvalidArgumentsException constructor initializes the exception message correctly.** **testConstructorNegative:** **Ensures that the InvalidArgumentsException constructor creates a non-null exception instance.**	
Input Value	
setUp: **exception:null** **testConstructorPositive:** **expectedMessage: "Invalid argument." (String)** **testConstructorNegative:**	
Expected Value	**Actual Value**
notNull assertions validate being not null.	**Result is as expected.**

Result of Test Case	***successful***
Test Script	

// The testConstructorPositive method creates an InvalidArgumentsException instance,
// verifying that the message is initialized correctly.
 	
// Create the exception instance using the constructor
exception = new InvalidArgumentsException();
 	
// Verify that the message is formatted correctly
String expectedMessage = "Invalid argument.";
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
	// The testConstructorNegative method checks the behavior of InvalidArgumentsException
	// when the message is not the expected one, which is not quite applicable here,
	// but we can test the exception instance for being non-null.
	
	// Create the exception instance using the constructor
	exception = new InvalidArgumentsException();
	
	// Verify that the exception is not null
	assertNotNull("Exception instance should not be null", exception);

Test Case 7: InvalidUserCredentialsExceptionTest	
Test Definition	
We are testing if the exception returns back the correct message.	
Input Value	
Positive: “Invalid user credentials” Negative: “User credentials are invalid”	
Expected Value	**Actual Value**
Invalid user credentials	Invalid user credentials”
Result of Test Case	Successfull
Test Script	

@Test
public void testConstructorMessageNotEquals() {
// The testConstructorMessageNotEquals method creates an InvalidUserCredentialsException instance,
// verifying that the message does not match an incorrect message.
 	
// Create the exception instance using the constructor
exception = new InvalidUserCredentialsException();
	@Test
	public void testConstructorMessageNotEquals() {
	// The testConstructorMessageNotEquals method creates an InvalidUserCredentialsException instance,
	// verifying that the message does not match an incorrect message.
	
	// Create the exception instance using the constructor
	exception = new InvalidUserCredentialsException();
	
	// Verify that the message is not equal to an

// Verify that the message is not equal to an incorrect message
String incorrectMessage = "User credentials are invalid";
assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
}
	incorrect message
	String incorrectMessage = "User credentials are invalid";
	assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
	}

Test Case 8: PasswordsDoNotMatchExceptionTest	
Test Definition	
We are checking if the exception gives out the correct exception message.	
Input Value	

Positive: exception=new PasswordsDoNotMatchException();
Negative: exception=new PasswordsDoNotMatchException();	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	***Successfull***
Test Script	

@Test
public void testConstructorMessageEquals() {
// The testConstructorMessageEquals method creates a PasswordsDoNotMatchException instance,
// verifying that the message is initialized correctly.
 	
// Create the exception instance using the constructor
exception = new PasswordsDoNotMatchException();
 	
// Verify that the message is formatted
	@Test
	public void testConstructorMessageNotEquals() {
	// The testConstructorMessageNotEquals method creates a PasswordsDoNotMatchException instance,
	// verifying that the message does not match an incorrect message.
	
	// Create the exception instance using the constructor
	exception = new PasswordsDoNotMatchException();

correctly
String expectedMessage = "The Password and Repeat password fields do not match.";
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
}
	// Verify that the message is not equal to an incorrect message
	String incorrectMessage = "Passwords do not match.";
	assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
	}

Test Case 9: ProductAlreadyInShoppingCartException	
Test Definition	
Checking if the exception gives out the correct message	
Input Value	

Long productId = 123L;
String username = "testUser";	

Expected Value	**Actual Value**
Equals assertion validates equality of messages. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

@Test
public void testConstructorMessageEquals() {
// The testConstructorMessageEquals method creates a ProductAlreadyInShoppingCartException instance,
// verifying that the message is initialized correctly.
 	
// Define the product ID and username that will be used to construct the exception
Long productId = 123L;
String username = "testUser";
	@Test
	public void testConstructorMessageNotEquals() {
	// The testConstructorMessageNotEquals method creates a ProductAlreadyInShoppingCartException instance,
	// verifying that the message does not match an incorrect message.
	
	// Define the product ID and username that will be used to construct the exception
	Long productId = 123L;

// Create the exception instance using the constructor
exception = new ProductAlreadyInShoppingCartException(productId, username);
 	
// Verify that the message is formatted correctly
String expectedMessage = String.format("Product with id: %d already exists in shopping cart for user with username %s", productId, username);
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
}
	String username = "testUser";
	
	// Create the exception instance using the constructor
	exception = new ProductAlreadyInShoppingCartException(productId, username);
	
	// Verify that the message is not equal to an incorrect message
	String incorrectMessage = "Product is already in the shopping cart.";
	assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
	}

Test Case 10: ProductNotFoundExceptionTest	
Test Definition	
Checking if the exception gives out the correct message.	
Input Value	

Long productId = 123L;
 	
exception = new ProductNotFoundException(productId);	

Expected Value	**Actual Value**
Equals assertion validates equality of messages. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

@Test
public void testConstructorPositive() {
// The testConstructorPositive method creates a
	@Test
	public void testConstructorNegative() {
	// The testConstructorNegative method creates a ProductNotFoundException instance with a null

ProductNotFoundException instance with a valid productId,
// verifying that the message is initialized correctly.
 	
// Define the product ID that will be used to construct the exception
Long productId = 123L;
 	
// Create the exception instance using the constructor
exception = new ProductNotFoundException(productId);
 	
// Verify that the message is formatted correctly
String expectedMessage = String.format("Product with id: %d was not found", productId);
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
}
	productId,
	// checking if the message is handled correctly. We assume that the constructor should handle nulls gracefully.
	
	// Define a null product ID
	Long productId = null;
	
	// Create the exception instance using the constructor
	exception = new ProductNotFoundException(productId);
	
	// Verify that the message is handled correctly
	String expectedMessage = "Product with id: null was not found";
	assertEquals("Exception message should handle null ID", expectedMessage, exception.getMessage());
	}

Test Case 11: ShoppingCartNotFoundException	
Test Definition	
Checking if the exception gives out the correct message.	
Input Value	

Long cartId = 123L;
 	
exception = new ShoppingCartNotFoundException(cartId);	

Expected Value	**Actual Value**
Equals assertion validates equality of messages. **False assertion validates inequality.**	Result is as expected.
Result of Test Case	Successfull

Test Script	

@Test
public void testConstructorPositive() {
// The testConstructorPositive method creates a ShoppingCartNotFoundException instance with a valid cartId,
// verifying that the message is initialized correctly.
 	
// Define the cart ID that will be used to construct the exception
Long cartId = 123L;
 	
// Create the exception instance using the constructor
exception = new ShoppingCartNotFoundException(cartId);
 	
// Verify that the message is formatted correctly
String expectedMessage = String.format("Shopping cart with id: %d was not found", cartId);
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
 	
// Verify that the message is not equal to an incorrect message
String incorrectMessage = "Shopping cart not found";
assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
}
	@Test
	public void testConstructorNegative() {
	// The testConstructorNegative method creates a ShoppingCartNotFoundException instance with a null cartId,
	// checking if the message is handled correctly. We assume that the constructor should handle nulls gracefully.
	
	// Define a null cart ID
	Long cartId = null;
	
	// Create the exception instance using the constructor
	exception = new ShoppingCartNotFoundException(cartId);
	
	// Verify that the message is not equal to an incorrect message
	String incorrectMessage = "Shopping cart not found";
	assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
	}

Test Case 12: UserNotFoundExceptionTest	
Test Definition	
Seeing if the exception gives out the correct message.	
Input Value	
String username = "testUser"; exception = new UserNotFoundException(username);	
Expected Value	**Actual Value**
Equals assertion validates equality of messages. False assertion validates inequality.	Result is as expected.
Result of Test Case	Successful
Test Script	

```
@Test
    public void testConstructorMessageEquals() {
        // The testConstructorMessageEquals method creates a UserNotFoundException instance,
        // verifying that the message is initialized correctly.

        // Define the username that will be used to construct the exception
        String username = "testUser";

        // Create the exception instance using the constructor
        exception = new UserNotFoundException(username);

        // Verify that the message is formatted correctly
        String expectedMessage = String.format("User with username: %s was not found", username);
        assertEquals("Exception message should match", expectedMessage, exception.getMessage());
    }
```

```
@Test
    public void testConstructorMessageNotEquals() {
        // The testConstructorMessageNotEquals method creates a UserNotFoundException instance,
        // verifying that the message does not match an incorrect message.

        // Define the username that will be used to construct the exception
        String username = "testUser";

        // Create the exception instance using the constructor
        exception = new UserNotFoundException(username);

        // Verify that the message is not equal to an incorrect message
        String incorrectMessage = "User not found.";
        assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
```

	}
Test Case 13: UsernameAlreadyExistsExceptionTest	
Test Definition	
Checking if the exception gives out the correct message.	
Input Value	

String username = "testUser";
 	
exception = new UsernameAlreadyExistsException(username);	

Expected Value	**Actual Value**
Equals assertion validates equality of messages. **False assertion validates inequality.**	Result is as expected
Result of Test Case	Successfull
Test Script	

< @Test
public void testConstructorMessageEquals() {
// The testConstructorMessageEquals method creates a UsernameAlreadyExistsException instance,
// verifying that the message is initialized correctly.
 	
// Define the username that will be used to construct the exception
String username = "testUser";
 	
// Create the exception instance using the constructor
exception = new UsernameAlreadyExistsException(username);
 	
// Verify that the message is formatted correctly
String expectedMessage =
	@Test
	public void testConstructorMessageNotEquals() {
	// The testConstructorMessageNotEquals method creates a UsernameAlreadyExistsException instance,
	// verifying that the message does not match an incorrect message.
	
	// Define the username that will be used to construct the exception
	String username = "testUser";
	
	// Create the exception instance using the constructor
	exception = new UsernameAlreadyExistsException(username);
	
	// Verify that the message is not equal to an incorrect message

String.format("User with username: %s already exists", username);
assertEquals("Exception message should match", expectedMessage, exception.getMessage());
}
	String incorrectMessage = "Username already exists.";
	assertNotEquals("Exception message should not match the incorrect message", incorrectMessage, exception.getMessage());
	}

Test Case 14:ProductServiceImplDeleteByIdTest	
Test Definition	
Testing the Product Service Implementation if it deletes a product by Id correctly.	
Input Value	

Long productId = 1L;
Product existingProduct = new Product("Old Product Name", 10.0, 5, new Category(), new Manufacturer());	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

@Test
void testDeleteById() {
Long productId = 1L;
Product existingProduct = new Product("Old Product Name", 10.0, 5, new Category(), new Manufacturer());
 	
when(productRepository.findById(productId)).thenReturn(Optional.of(existingProduct));
 	
productService.deleteById(productId);
 	
verify(productRepository, times(1)).deleteById(productId);
	@Test
	void testDeleteByIdNegative() {
	Long productId = 1L;
	
	when(productRepository.findById(productId)).thenReturn(Optional.empty());
	
	Optional<Product> deletedProduct = productService.findById(productId);
	
	assertFalse(deletedProduct.isPresent());
	assertNotEquals(productId, deletedProduct.map(Product::getId).orElse(null));

}	}
Test Case 15: ProductServiceImplEditTest	
Test Definition	
Checking if the Product Service Impl edits a product correctly	
Input Value	

Long productId = 1L;
String newName = "New Product Name";
Double newPrice = 20.0;
Integer newQuantity = 8;
Long categoryId = 1L;
Long manufacturerId = 1L;
Product existingProduct = new Product("Old Product Name", 10.0, 5, new Category(), new Manufacturer());
Category newCategory = new Category();
Manufacturer newManufacturer = new Manufacturer();	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

**@Test
void testEdit() {
Long productId = 1L;
String newName = "New Product Name";
Double newPrice = 20.0;
Integer newQuantity = 8;
Long categoryId = 1L;
Long manufacturerId = 1L;
Product existingProduct = new Product("Old Product Name", 10.0, 5, new Category(), new Manufacturer());
 	
when(productRepository.findById(productId))**
	@Test
	void testEditNegative() {
	Long productId = 1L;
	String newName = "New Product Name";
	Double newPrice = 20.0;
	Integer newQuantity = 8;
	Long categoryId = 1L;
	Long manufacturerId = 1L;
	Product existingProduct = new Product("Old Product Name", 10.0, 5, new Category(), new Manufacturer());

.thenReturn(Optional.of(existingProduct));
Category newCategory = new Category();
 	
when(categoryRepository.findById(categoryId)).thenReturn(Optional.of(newCategory));
Manufacturer newManufacturer = new Manufacturer();
 	
when(manufacturerRepository.findById(manufacturerId)).thenReturn(Optional.of(newManufacturer));
 	
Optional<Product> result = productService.edit(productId, newName, newPrice, newQuantity, categoryId, manufacturerId);
 	
assertTrue(result.isPresent());
assertEquals(newName, result.get().getName());
assertEquals(newPrice, result.get().getPrice());
assertEquals(newQuantity, result.get().getQuantity());
assertEquals(newCategory, result.get().getCategory());
assertEquals(newManufacturer, result.get().getManufacturer());
}
	when(productRepository.findById(productId)).thenReturn(Optional.of(existingProduct));
	Category newCategory = new Category();
	
	when(categoryRepository.findById(categoryId)).thenReturn(Optional.of(newCategory));
	Manufacturer newManufacturer = new Manufacturer();
	
	when(manufacturerRepository.findById(manufacturerId)).thenReturn(Optional.of(newManufacturer));
	
	Optional<Product> result = productService.edit(productId, newName, newPrice, newQuantity, categoryId, manufacturerId);
	
	assertNotEquals("newName", result.get().getName());
	assertNotEquals("newPrice", result.get().getPrice());
	assertNotEquals("newQuantity", result.get().getQuantity());
	assertNotEquals("newCategory", result.get().getCategory());
	assertNotEquals("newManufacturer", result.get().getManufacturer());
	}

Test Case 16: ProductServiceImplFindAllTest
Test Definition
Checking if the Product Service Impl does the FindAll method correctly
Input Value

List<Product> productList = new ArrayList<>();
productList.add(new Product("Product1", 10.0, 5, new Category(), new Manufacturer()));

productList.add(new Product("Product2", 15.0, 3, new Category(), new Manufacturer()));	
Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful.
Test Script	

@Test
  void testFindAll() {
    //Making a product list
    List<Product> productList = new ArrayList<>();
    productList.add(new Product("Product1", 10.0, 5, new Category(), new Manufacturer()));
    productList.add(new Product("Product2", 15.0, 3, new Category(), new Manufacturer()));
 	
when(productRepository.findAll()).thenReturn(productList);
 	
    //getting the result of the method findAll
    List<Product> result = productService.findAll();
 	
    //seeing if the result is equal
    assertEquals(2, result.size());
  }
	@Test
	  void testFindAllNegative() {
	    //Making a product list
	    List<Product> productList = new ArrayList<>();
	    productList.add(new Product("Product1", 10.0, 5, new Category(), new Manufacturer()));
	    productList.add(new Product("Product2", 15.0, 3, new Category(), new Manufacturer()));
	
	when(productRepository.findAll()).thenReturn(productList);
	
	    //getting the result of the method findAll
	    List<Product> result = productService.findAll();
	
	    //seeing if the result is equal
	    assertNotEquals(3, result.size());
	  }

Test Case 17: ProductServiceImplFindByIdTest	
Test Definition	
-||- findById method	
Input Value	

Long productId = 1L;
    Product product = new Product("Test Product", 10.0, 5, new Category(), new Manufacturer());	

Expected Value	**Actual Value**

Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

```
@Test
   void testFindById() {



       Long productId = 1L;
       Product product = new Product("Test Product", 10.0, 5, new Category(), new Manufacturer());

when(productRepository.findById(productId)).thenReturn(Optional.of(product));



       Optional<Product> result = productService.findById(productId);


       assertEquals(product, result.orElseThrow()); // Ensure the product returned matches the one we expect
   }
```

```
@Test
   void testFindByIdNegative() {
       Long productId = 1L;
       Product notExpectedProduct = new Product("Test Product", 10.0, 5, new Category(), new Manufacturer());

when(productRepository.findById(productId)).thenReturn(Optional.empty());


       Optional<Product> result = productService.findById(productId);


       assertFalse(result.isPresent()); // Ensure no product is returned
       assertNotEquals(notExpectedProduct, result.orElse(null)); // Ensure the returned product is not the unexpected one
   }
```

Test Case 18: ProductServiceImplFindByNameTest	
Test Definition	
-||- findByName method	
Input Value	

String productName = "Test Product";
Product product = new Product(productName, 10.0, 5, new Category(), new Manufacturer());	

Expected Value	**Actual Value**
Equals assertion validates equality.	Result is as expected

False assertion validates inequality.	
Result of Test Case	Successfull
Test Script	

< @Test
    void testFindByName_Success() {
        String productName = "Test Product";
        Product product = new Product(productName, 10.0, 5, new Category(), new Manufacturer());
 	
when(productRepository.findByName(productName)).thenReturn(Optional.of(product));
 	
        Optional<Product> result = productService.findByName(productName);
 	
        assertEquals(product, result.orElseThrow()); // Ensure the correct product is returned
    }
	@Test
	    void testFindByName_ProductNotFound() {
	        String productName = "Nonexistent Product";
	
	when(productRepository.findByName(productName)).thenReturn(Optional.empty());
	
	    }

Test Case 19: ProductServiceImplSaveTest	
Test Definition	
-||- Save method	
Input Value	

String productName = "Test Product";
        Double productPrice = 10.0;
        Integer productQuantity = 5;
        Long categoryId = 1L;
        Long manufacturerId = 1L;
        Category category = new Category();
Manufacturer manufacturer = new Manufacturer();
Product savedProduct = new Product(productName, productPrice, productQuantity, category, manufacturer);	

Expected Value	**Actual Value**

Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

@Test
void testSave_Success() {
String productName = "Test Product";
Double productPrice = 10.0;
Integer productQuantity = 5;
Long categoryId = 1L;
Long manufacturerId = 1L;
Category category = new Category();
 	
when(categoryRepository.findById(categoryId)).thenReturn(Optional.of(category));
Manufacturer manufacturer = new Manufacturer();
 	
when(manufacturerRepository.findById(manufacturerId)).thenReturn(Optional.of(manufacturer));
Product savedProduct = new Product(productName, productPrice, productQuantity, category, manufacturer);
 	
when(productRepository.save(any(Product.class))).thenReturn(savedProduct);
 	
Optional<Product> result = productService.save(productName, productPrice, productQuantity, categoryId, manufacturerId);
 	
assertTrue(result.isPresent());
assertEquals(savedProduct, result.get());
}
	@Test
	void testSave_CategoryNotFound() {
	String productName = "Test Product";
	Double productPrice = 10.0;
	Integer productQuantity = 5;
	Long categoryId = 1L;
	Long manufacturerId = 1L;
	
	when(categoryRepository.findById(categoryId)).thenReturn(Optional.empty());
	
	CategoryNotFoundException exception = null;
	try {
	productService.save(productName, productPrice, productQuantity, categoryId, manufacturerId).orElseThrow();
	} catch (CategoryNotFoundException e) {
	exception = e;
	}
	
	assertNotNull(exception);
	assertNotEquals(null, exception.getMessage());
	}
	
	@Test
	void testSave_ManufacturerNotFound() {
	String productName = "Test Product";
	Double productPrice = 10.0;
	Integer productQuantity = 5;
	Long categoryId = 1L;
	Long manufacturerId = 1L;
	Category category = new Category();

	when(categoryRepository.findById(category Id)).thenReturn(Optional.of(category));
	
	when(manufacturerRepository.findById(ma nufacturerId)).thenReturn(Optional.empty());
	
	    // When
	    ManufacturerNotFoundException exception = null;
	    try {
	      productService.save(productName, productPrice, productQuantity, categoryId, manufacturerId).orElseThrow();
	    } catch (ManufacturerNotFoundException e) {
	      exception = e;
	    }
	
	    assertNotNull(exception);
	    assertNotEquals(null, exception.getMessage());
	  }

Test Case 20: ShoppingCartConstructorTest	
Test Definition	
Checking if the ShoppingCartConstructor works as intended.	
Input Value	

user = new User("username", "password", "John", "Doe", null, null);
    shoppingCart = new ShoppingCart(user);	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	**Result is as expected**
Result of Test Case	Successfull
Test Script	

**public void testShoppingCartConstructor() {
    assertEquals(user, shoppingCart.getUser());**
	public void testShoppingCartConstructorNegative() {
	    User anotherUser = new User("anotherUsername", "password",

assertEquals(ShoppingCartStatus.CREATED, shoppingCart.getStatus());
   }
	"Jane", "Doe", null, null);
	        assertNotEquals(anotherUser, shoppingCart.getUser());
	
	assertNotEquals(ShoppingCartStatus.CANCELED, shoppingCart.getStatus());
	   }

Test Case 21: ShoppingCartDataUpdateTest	
Test Definition	
Checking if the update method works as intended.	
Input Value	

user = new User("username", "password", "John", "Doe", null, null);
        shoppingCart = new ShoppingCart(user);	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

< public void testUpdateShoppingCartData() {
        User newUser = new User("newUsername", "newPassword", "Jane", "Doe", null, null);
        shoppingCart.setUser(newUser);
 	
shoppingCart.setStatus(ShoppingCartStatus.CREATED);
 	
        assertEquals(newUser, shoppingCart.getUser());
 	
assertEquals(ShoppingCartStatus.CREATED, shoppingCart.getStatus());
   }
	public void testUpdateShoppingCartDataNegative() {
	        User anotherUser = new User("anotherUsername", "password", "Jane", "Doe", null, null);
	        shoppingCart.setUser(anotherUser);
	
	shoppingCart.setStatus(ShoppingCartStatus.CREATED);
	
	        assertNotEquals(user, shoppingCart.getUser());
	
	assertNotEquals(ShoppingCartStatus.CANCELED, shoppingCart.getStatus());
	   }

Test Case 22: ProductConstructorTest	
Test Definition	
Checking if the constructor works as intended	
Input Value	

category = new Category("Electronics","elec");
manufacturer = new Manufacturer("El Comp.","mach");
product = new Product("Laptop", 999.99, 10, category, manufacturer);	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

public void testProductConstructor() {
 	
assertEquals("Laptop", product.getName());
assertEquals(999.99, product.getPrice());
assertEquals(10,(int)product.getQuantity());
assertEquals(category, product.getCategory());
assertEquals(manufacturer, product.getManufacturer());
}
	public void testProductConstructorNegative(){
	Category category1= new Category("smth","smth");
	Manufacturer manufacturer1=new Manufacturer("msmth","msmth");
	assertNotEquals("Laptop1", product.getName());
	assertNotEquals(1000d, product.getPrice());
	assertNotEquals(11,(int)product.getQuantity());
	assertNotEquals(category1, product.getCategory());
	assertNotEquals(manufacturer1, product.getManufacturer());
	}

Test Case 23: GetHomePageTest	
Test Definition	
Checking if the url for homepage is correct	
Input Value	

MockitoAnnotations.initMocks(this);
homeController = new HomeController();	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

**@Test
public void testGetHomePagePositive() {
// Mock behavior of Model
 	
when(model.addAttribute("bodyContent", "home")).thenReturn(model);
 	
// Call the getHomePage method
String result = homeController.getHomePage(model);
 	
// Assert that the returned view name is "master-template"
assertEquals("master-template", result);
}**
	@Test
	public void testGetHomePageNegative() {
	// Mock behavior of Model (no need to configure for this test)
	
	// Call the getHomePage method
	String result = homeController.getHomePage(model);
	
	// Assert that the returned view name is not "invalid-template"
	assertNotEquals("invalid-template", result);
	}

Test Case 24: GetLoginPageTest	
Test Definition	
Checking if we get the url for Login Page	
Input Value	

loginController = new LoginController(null); // AuthService isn't used in getLoginPage, so we can pass null
model = new ConcurrentModel(); // Model for storing attributes	

Expected Value	**Actual Value**

Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful
Test Script	
@Test public void testGetLoginPagePositive() { // Call the getLoginPage method String viewName = loginController.getLoginPage(model); // Verify the expected view name is returned assertEquals("master-template", viewName); // Check if the expected model attribute is set assertTrue(model.containsAttribute("bodyContent")); assertEquals("login", model.asMap().get("bodyContent")); }	@Test public void testGetLoginPageNegative() { // Check that the returned view name is not incorrect String viewName = loginController.getLoginPage(model); // Make sure it does not return an unexpected view name assertNotEquals("wrong-template", viewName); // Ensure that model's 'bodyContent' is not set to something unexpected assertNotEquals("error", model.asMap().get("bodyContent")); // Additional checks assertNotNull(viewName); // Make sure the view name is not null assertNotNull(model.asMap().get("bodyContent")); // Ensure 'bodyContent' is set // The expected positive result should still hold assertEquals("master-template", viewName); assertEquals("login", model.asMap().get("bodyContent")); }
Test Case 25 : LogoutTest	
Test Definition	
Getting logout page correctly	
Input Value	

logoutController = new LogoutController();
        request = Mockito.mock(HttpServletRequest.class);
        session = Mockito.mock(HttpSession.class); // Create a mocked session
        Mockito.when(request.getSession()).thenReturn(session); // Return this mocked session
        model = new ConcurrentModel(); // Model to pass into controller methods
String result = logoutController.logout(request, model);	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successfull
Test Script	

@Test
public void testLogoutPositive() {
String result = logoutController.logout(request, model);
 	
// Verify the session was invalidated
Mockito.verify(session, Mockito.times(1)).invalidate(); // Ensure session is invalidated once
 	
// Check the expected redirection
assertEquals("redirect:/login", result);
}
	@Test
	public void testLogoutNegative() {
	String result = logoutController.logout(request, model);
	
	// Verify the session was invalidated
	Mockito.verify(session, Mockito.times(1)).invalidate(); // Ensure session is invalidated once
	
	// Check the expected redirection
	assertNotEquals("redirect:/invalid-template", result);
	}

Test Case 26: addProductPageTest
Test Definition
Are we getting the correct url for the addProductPage
Input Value

ProductService productService = mock(ProductService.class);
        CategoryService categoryService = mock(CategoryService.class);
        ManufacturerService manufacturerService = mock(ManufacturerService.class);

model = mock(Model.class); productController = new ProductController(productService, categoryService, manufacturerService);	
Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	***<successful OR fail>***

public void testAddProductPage() {
 	
 // Preparing test data
 List<Manufacturer> manufacturers = new ArrayList<>();
 manufacturers.add(new Manufacturer("Manufacturer 1","man1"));
 manufacturers.add(new Manufacturer("Manufacturer 2","man2"));
 List<Category> categories = new ArrayList<>();
 categories.add(new Category("Category 1","cat1"));
 categories.add(new Category("Category 2","cat2"));
 	
 // Call the method under test
 String viewName = productController.addProductPage(model);
 	
 // Assert the view name
 assertEquals("View name should be 'master-template", "master-template", viewName);
 }
	public void testAddProductPageNegative() {
	
	 // Preparing test data
	 List<Manufacturer> manufacturers = new ArrayList<>();
	 manufacturers.add(new Manufacturer("Manufacturer 1","man1"));
	 manufacturers.add(new Manufacturer("Manufacturer 2","man2"));
	 List<Category> categories = new ArrayList<>();
	 categories.add(new Category("Category 1","cat1"));
	 categories.add(new Category("Category 2","cat2"));
	
	 // Call the method under test
	 String viewName = productController.addProductPage(model);
	
	 // Assert the view name
	 assertNotEquals("invalid-template", viewName,"View name should not be invalid-template");
	 }

<Put code here>	
Test Case 27: DeleteProductTest	

Test Definition	
Are we getting the correct url after deleting a product	
Input Value	

productService=mock(ProductService.class);
model=mock(Model.class);
productController=new ProductController(productService,null,null);	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful
Test Script	

public void testDeleteProduct() {
 	
// Calling the method under test
String viewName = productController.deleteProduct(123L);
 	
// Asserting the view name
assertEquals("View name should be 'redirect:/products' after deletion","redirect:/products", viewName);
}
	public void testDeleteProductNegative() {
	// Calling the method under test
	String viewName = productController.deleteProduct(123L);
	
	// Asserting the view name is not equal to "redirect:/products" after deletion
	assertNotEquals("redirect:/products/{id}", viewName,"View name should not be 'redirect:/products/{id} after deletion");
	}

Test Case 28: editProductTest
Test Definition
Are we getting the right url after editing a product
Input Value

productService=mock(ProductService.class);
model=mock(Model.class);

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful
Test Script	

public void testEditProduct() {
// Mocking ProductService to return a dummy product
 	
when(productService.findById(1L)).thenReturn(Optional.of(new Product()));
 	
// Creating ProductController with mocked ProductService
productController = new ProductController(productService, null, null);
 	
// Calling the method under test
String viewName = productController.editProduct(1L, "Test Product", 50.0, 5, 1L, 1L);
 	
// Asserting that the view name is "redirect:/products"
assertEquals("View name should be 'master-template' after successfully editing product", viewName, "redirect:/products");
}
	public void testEditProductNegative() {
	// Mocking ProductService to return empty Optional, indicating product not found
	
	when(productService.findById(1L)).thenReturn(Optional.empty());
	
	// Creating ProductController with mocked ProductService
	productController = new ProductController(productService, null, null);
	
	// Calling the method under test
	String viewName = productController.editProduct(1L, "Test Product", 50.0, 5, 1L, 1L);
	
	// Asserting that the view name is "redirect:/products?error=ProductNotFound"
	assertNotEquals("View name should not be invalid-template", "redirect:/products", viewName);
	}

Test Case 29: registerTest	
Test Definition	
Are we getting the correct url after registering	
Input Value	

String username = "testUser";
String password = "testPassword";	

String repeatedPassword = "testPassword";
String name = "John";
String surname = "Doe";
Role role = Role.ROLE_USER;	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful
Test Script	

@Test
public void testRegisterPositive() {
// Arrange
String username = "testUser";
String password = "testPassword";
String repeatedPassword = "testPassword";
String name = "John";
String surname = "Doe";
Role role = Role.ROLE_USER;
 	
// Act
String result = registerController.register(username, password, repeatedPassword, name, surname, role);
 	
// Assert
assertEquals("redirect:/login", result);
// Additional assertions can be made to verify the state of the system after registration
}
	@Test
	public void testRegisterNegative() {
	// Arrange
	String username = ""; // Empty username
	String password = "testPassword";
	String repeatedPassword = "testPassword";
	String name = "John";
	String surname = "Doe";
	Role role = Role.ROLE_USER;
	
	// Act
	String result = registerController.register(username, password, repeatedPassword, name, surname, role);
	
	// Assert
	
	assertNotEquals("redirect:/register?error=Invalid%20username.", result);
	}

Test Case 30: getManufacturersPageTest
Test Definition

Are we getting the right url for the manufacturers	
Input Value	

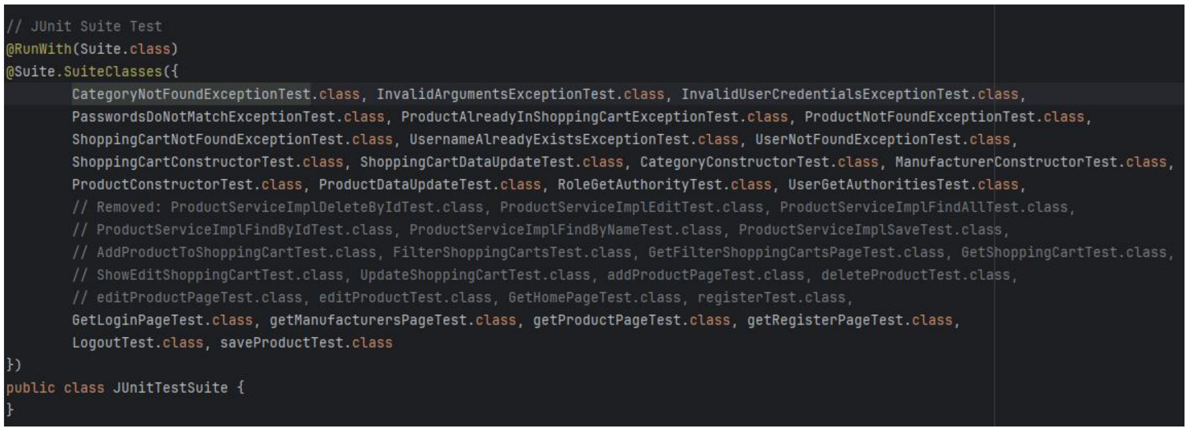
mockManufacturerService = mock(ManufacturerService.class);
        controller = new ManufacturerController(mockManufacturerService);
        model = new ConcurrentModel();	

Expected Value	**Actual Value**
Equals assertion validates equality. False assertion validates inequality.	Result is as expected
Result of Test Case	Successful
Test Script	

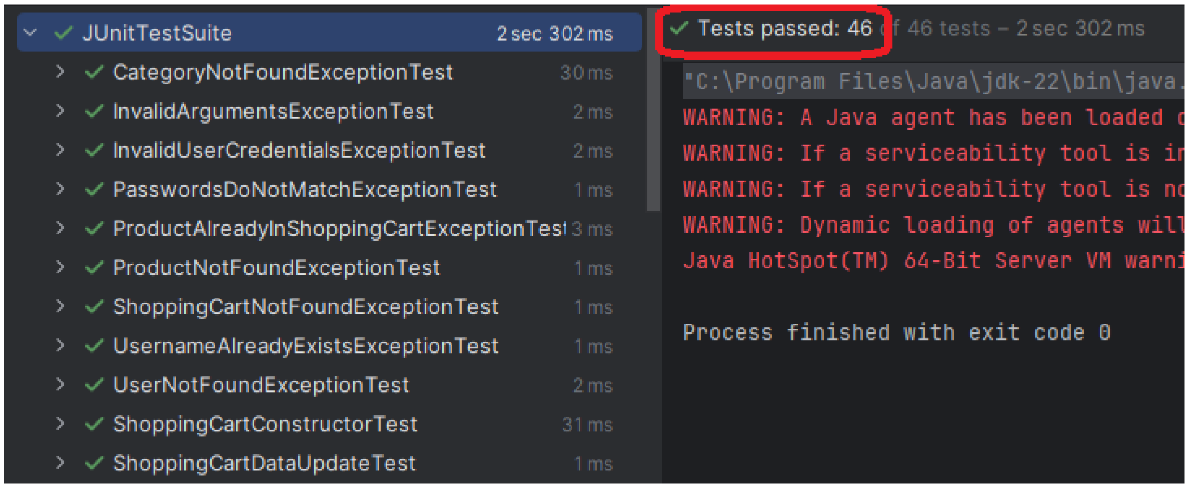
**@Test
    public void testGetManufacturersPage_Positive() {
        // Creating some sample manufacturers
        List<Manufacturer> sampleManufacturers = new ArrayList<>();
        sampleManufacturers.add(new Manufacturer("Manufacturer A", "UK"));
        sampleManufacturers.add(new Manufacturer("Manufacturer B", "USA"));
 	
        // Stubbing the findAll method to return the sample manufacturers
        when(mockManufacturerService.findAll()).thenReturn(sampleManufacturers);
 	
        // Calling the method under test
        String viewName = controller.getManufacturersPage(model);
 	
        // Asserting that the model contains the manufacturers attribute
        assertTrue(model.containsAttribute("manufacturers"));
 	
        // Asserting that the view name returned is correct
        assertEquals("master-template", viewName);
    }**
	@Test
	    public void testGetManufacturersPage_Negative() {
	        // Stubbing the findAll method to return null (simulating an empty result)
	        when(mockManufacturerService.findAll()).thenReturn(null);
	
	        // Calling the method under test
	        String viewName = controller.getManufacturersPage(model);
	
	        // Asserting that the model does not contain the manufacturers attribute
	        assertFalse(model.containsAttribute("manufacturers"));
	
	        // Asserting that the view name returned is correct
	        assertNotEquals("invalid-template", viewName);
	    }

4. CONCLUSION

In conclusion, we worked on the project as a team, working on different options to make the project look as good as possible. We thought about different test cases which we might want to and might not want to do for the needed requirements. After everyone finished up their part, we ended up with 41 test cases, if we have not made a mistake counting them.
Finally, with 3'rd version of the project, there are (46/2) 23 positive-negative successfull test pairs and 18 failed pairs as left commented as its seen the following picture.



```java
// JUnit Suite Test
@RunWith(Suite.class)
@Suite.SuiteClasses({
        CategoryNotFoundExceptionTest.class, InvalidArgumentsExceptionTest.class, InvalidUserCredentialsExceptionTest.class,
        PasswordsDoNotMatchExceptionTest.class, ProductAlreadyInShoppingCartExceptionTest.class, ProductNotFoundExceptionTest.class,
        ShoppingCartNotFoundExceptionTest.class, UsernameAlreadyExistsExceptionTest.class, UserNotFoundExceptionTest.class,
        ShoppingCartConstructorTest.class, ShoppingCartDataUpdateTest.class, CategoryConstructorTest.class, ManufacturerConstructorTest.class,
        ProductConstructorTest.class, ProductDataUpdateTest.class, RoleGetAuthorityTest.class, UserGetAuthoritiesTest.class,
        // Removed: ProductServiceImplDeleteByIdTest.class, ProductServiceImplEditTest.class, ProductServiceImplFindAllTest.class,
        // ProductServiceImplFindByIdTest.class, ProductServiceImplFindByNameTest.class, ProductServiceImplSaveTest.class,
        // AddProductToShoppingCartTest.class, FilterShoppingCartsTest.class, GetFilterShoppingCartsPageTest.class, GetShoppingCartTest.class,
        // ShowEditShoppingCartTest.class, UpdateShoppingCartTest.class, addProductPageTest.class, deleteProductTest.class,
        // editProductPageTest.class, editProductTest.class, GetHomePageTest.class, registerTest.class,
        GetLoginPageTest.class, getManufacturersPageTest.class, getProductPageTest.class, getRegisterPageTest.class,
        LogoutTest.class, saveProductTest.class
})
public class JUnitTestSuite {
}
```



```
✓ JUnitTestSuite                              2 sec 302 ms
  ✓ CategoryNotFoundExceptionTest                    30 ms
  ✓ InvalidArgumentsExceptionTest                     2 ms
  ✓ InvalidUserCredentialsExceptionTest               2 ms
  ✓ PasswordsDoNotMatchExceptionTest                  1 ms
  ✓ ProductAlreadyInShoppingCartExceptionTest         3 ms
  ✓ ProductNotFoundExceptionTest                      1 ms
  ✓ ShoppingCartNotFoundExceptionTest                 1 ms
  ✓ UsernameAlreadyExistsExceptionTest                1 ms
  ✓ UserNotFoundExceptionTest                         2 ms
  ✓ ShoppingCartConstructorTest                      31 ms
  ✓ ShoppingCartDataUpdateTest                        1 ms
```

```
✓ Tests passed: 46 of 46 tests – 2 sec 302 ms

"C:\Program Files\Java\jdk-22\bin\java.
WARNING: A Java agent has been loaded d
WARNING: If a serviceability tool is in
WARNING: If a serviceability tool is no
WARNING: Dynamic loading of agents will
Java HotSpot(TM) 64-Bit Server VM warni

Process finished with exit code 0
```