TEST CASE #1 (Dejvi Kocilja)

1. Introduction to Testing:

Software testing is a methodical procedure that is essential to assessing the Bill class and making sure it performs as planned. We check the code for flaws, faults, and defects by running the methods of the Bill class with different inputs and scenarios. Maintaining the accuracy, consistency, and caliber of the software application's billing functionality requires testing the Bill class.

2. Purpose of Testing:

Finding and fixing bugs early in the development cycle is the main goal of testing the Bill class. Early problem detection reduces the amount of money and time needed to repair problems. Furthermore, testing makes sure the Bill class works as intended—that is, that totals are calculated correctly, discounts are applied, and different billing circumstances are handled well. This eventually results in a better user experience by preserving the software's dependability and effectiveness.

3. Focus on Testing a Single Component:

Focusing on testing the Bill class allows us to thoroughly evaluate its role, complexity, and impact on the overall software system. Billing is a component with critical functionality, for which rigorous testing is necessary to ensure it is reliable and correct. Potential flaws or problems unique to billing computations and discount application can cause errors in calculating the profits or losses of the supermarket, which would in turn disrupt its overall activities. That is why it is of greatest importance to test the Bill class which handles the functions mentioned till now.

4. Preparing Test Cases:

Test cases for the Bill class should encompass a wide range of scenarios, including normal inputs (valid bill data), edge cases (empty bill, zero prices), and invalid inputs (non-numeric quantities, negative prices). Each test case should specify the inputs, actions, and expected outcomes, ensuring comprehensive coverage of the Bill class's functionality.

5. Choosing Testing Frameworks:

For testing the Bill class in Java, JUnit is a suitable testing framework. JUnit provides a structured approach to writing, organizing, and executing tests, offering features such as assertion libraries and test runners. By utilizing JUnit, we can efficiently set up and execute test cases for the Bill class, facilitating thorough testing of its methods and functionalities.

6. Writing Test Code:

Writing test code for the Bill class involves creating test methods to exercise different functionalities, such as calculating total bill amounts and applying discounts. In the piece of code below, I have first imported all the necessary libraries and created a function to initialize objects or perform necessary setup steps before executing individual test methods for the Bill class. Then function testTotalBillWithoutDiscount verifies that the total bill amount calculated by the totalBill method of the Bill class matches the expected value without applying any discount.

testTotalBillWithDiscount: Checks whether the total bill amount calculated by the totalBill method of the Bill class, after applying a 10% discount for a specific client, matches the expected discounted value.

testTotalBillWithEmptyBill: Ensures that the total bill amount for an empty bill (with no products) is calculated as 0.0 by the totalBill method of the Bill class.

testTotalBillWithInvalidInput: Validates that the totalBill method of the Bill class throws a NumberFormatException when attempting to calculate the total bill amount with an invalid input (negative price) for a product.

```
import static org.junit.Assert.*;
import java.util.ArrayList;
import org.junit.Before;
import org.junit.Test;

public class BillTest {
   private Bill bill;

   @Before
   public void setUp() {
      ArrayList<TextField> soldQnt = new ArrayList<>();
      soldQnt.add(new TextField("2"));
      soldQnt.add(new TextField("3"));
      ArrayList<Double> prices = new ArrayList<>();
      prices.add(10.0);
      prices.add(20.0);
      ArrayList<String> productsSold = new ArrayList<>();
      productsSold.add("Product1");
      productsSold.add("Product2");
      bill = new Bill("123", soldQnt, prices, "2024-05-03", productsSold);
   }
```

```
    @Test
    public void testTotalBillWithoutDiscount() {
        assertEquals(70.0, bill.totalBill(), 0.01);
    }

    @Test
    public void testTotalBillWithDiscount() {
        bill.setClient("Client1");
        bill.setDiscountPercentage(10);
        assertEquals(63.0, bill.totalBill(), 0.01);
    }
}
@Test
public void testTotalBillWithEmptyBill() {
    ArrayList<TextField> soldQnt = new ArrayList<>();
    ArrayList<Double> prices = new ArrayList<>();
    ArrayList<String> productsSold = new ArrayList<>();
    Bill emptyBill = new Bill("000", soldQnt, prices, "2024-05-03", productsSold);
    assertEquals(0.0, emptyBill.totalBill(), 0.01);
}

@Test(expected = NumberFormatException.class)
public void testTotalBillWithInvalidInput() {
    ArrayList<TextField> soldQnt = new ArrayList<>();
    soldQnt.add(new TextField("2"));
    ArrayList<Double> prices = new ArrayList<>();
    prices.add(-10.0); // Negative price
    ArrayList<String> productsSold = new ArrayList<>();
    productsSold.add("InvalidProduct");
```

```
Bill invalidBill = new Bill("999", soldQnt, prices, "2024-05-03", productsSold);

invalidBill.totalBill(); // This should throw NumberFormatException }
```

7. Running Tests:

Executing tests for the Bill class involves running the test code against its methods and analyzing the results. Tests may pass, fail, or encounter errors, indicating the correctness of the Bill class's implementation. By interpreting test results, we can identify any defects or deviations from expected behavior and take corrective actions as necessary.

testTotalBillWithoutDiscount: If the total bill amount calculated without applying any discount (bill.totalBill()) matches the expected value (70.0), the test will pass. Otherwise, it will fail, indicating a discrepancy in the calculation.

testTotalBillWithDiscount: If the total bill amount calculated after applying a 10% discount (bill.totalBill()) for a specific client matches the expected discounted value (63.0), the test will pass. Otherwise, it will fail, indicating a discrepancy in the calculation.

testTotalBillWithEmptyBill: If the total bill amount for an empty bill (emptyBill.totalBill()) is calculated as 0.0, the test will pass. Otherwise, it will fail, indicating that the totalBill method does not handle empty bills correctly.

testTotalBillWithInvalidInput: If the totalBill method throws a NumberFormatException as expected when attempting to calculate the total bill amount with an invalid input (negative price), the test will pass. If the exception is not thrown or a different exception occurs, the test will fail, indicating an unexpected behavior.

8. Test Coverage:

In order to thoroughly test the Bill class and make sure that the test suite is appropriately exercising all of its methods and features, achieving high test coverage is essential. Enough test coverage makes it easier to spot untested code paths and guarantees that the Bill class operates correctly and consistently across a range of billing scenarios. A more stable and dependable application can be produced by having a high test coverage level, which can aid in the early detection and correction of any possible flaws or difficulties. In summary, aiming for high test coverage in the Bill class is critical to preserving the billing system's overall integrity and quality.

TEST CASE #2 (Ifigjenia Sopiqoti)

1. Introduction to Testing:

Software testing is a fundamental process within the software development lifecycle aimed at identifying defects or bugs in software applications. It involves systematically executing the software to ensure it behaves as expected and fulfills specified requirements. Testing plays a pivotal role in software development by ensuring the reliability, correctness, and overall quality of the software product. By rigorously testing software components, developers can identify and address issues early in the development process, ultimately reducing costs and time spent on rework.

2. Purpose of Testing:

The primary purpose of testing is to detect defects early in the development process, minimizing the cost and effort required for rectifying them. Additionally, testing verifies that software components, such as the Inventory class, perform as intended and meet user needs. By testing software components thoroughly, developers can ensure that the final product is robust, reliable, and meets specified requirements.

3. Focus on Testing a Single Component:

The Inventory class plays a crucial role in managing the inventory of products within a software system. Testing this component is essential to ensure accurate inventory management and proper handling of product data. By focusing on testing the Inventory class, developers can identify potential defects or errors specific to inventory management functionalities, thereby ensuring the reliability and functionality of the overall system.

4. Preparing Test Cases:

Test cases serve as detailed specifications for testing specific functionalities or scenarios of the Inventory class. These cases should cover a range of scenarios, including normal operations, edge cases such as empty inventory or missing products, and invalid inputs such as non-existent products or negative quantities. By preparing comprehensive test cases, developers can ensure thorough testing of the Inventory class and identify potential issues across various scenarios.

5. Choosing Testing Frameworks:

In Java development, JUnit stands as a suitable testing framework for writing and executing tests, organizing test cases, and reporting results. Setting up the testing environment with JUnit involves adding the JUnit dependency to the project's build configuration. By leveraging JUnit, developers can streamline the testing process and ensure effective testing of the Inventory class.

6. Writing Test Code:

Writing test methods for the Inventory class involves creating tests to cover different functionalities such as adding products, updating stock, and checking for missing products. These test methods should be well-structured, readable, and maintainable, utilizing assertions to validate expected outcomes against actual results. By writing effective test code, developers can ensure thorough testing and verify the correctness of the Inventory class's behavior.

```
import static org.junit.Assert.*;
import java.util.ArrayList;
import org.junit.Before;
import org.junit.Test;

public class InventoryTest {
   private Inventory inventory;

   @Before
   public void setUp() {
      inventory = new Inventory();
   }

   @Test
   public void testAddProduct() {
      Product newProduct = new Product("TestProduct", 10.0, 10);
      inventory.addProduct(newProduct);
      assertTrue(inventory.isThere("TestProduct"));
   }

   @Test
   public void testUpdateStock() {
      inventory.addProduct(new Product("TestProduct", 10.0, 10));
      inventory.updateStock(5, "TestProduct");
      assertEquals(15, inventory.getProduct(inventory.getPosition("TestProduct")).getStock());
   }

   @Test
   public void testCheckForMissing() {
```

```
        inventory.addProduct(new Product("TestProduct1", 10.0, 3));
        inventory.addProduct(new Product("TestProduct2", 10.0, 7));
        ArrayList<String> missingProducts = inventory.checkForMissing();
        assertTrue(missingProducts.contains("Produkti : TestProduct1 has less than 5 pieces in stock"));
    }
}
```

7. Running Tests:

To execute the tests, you would typically run the InventoryTest class using a testing framework like JUnit. The framework would automatically initialize the Inventory object before each test method is executed, as specified by the @Before annotation on the setUp() method. Interpreting the results involves analyzing the outcomes of each test case. In the case of passing scenarios, assertions within test methods, such as assertTrue() or assertEquals(), verify expected behavior against actual behavior. If the actual behavior matches the expected behavior, the test case passes, indicating that the functionality being tested works as intended. For example, in the testAddProduct() method, the assertion assertTrue(inventory.isThere("TestProduct")) checks if the product was successfully added to the inventory. If the product is indeed present in the inventory, the test passes. In contrast, if the actual behavior deviates from the expected behavior, the test case fails. This could occur due to issues such as incorrect implementation or unexpected behavior within the code being tested.

8. Test Coverage:

High test coverage is essential to ensure thorough testing of the Inventory class, covering all its methods, edge cases, and interactions with product data. By achieving high test coverage, developers can identify areas of the code that are not adequately tested and ensure that critical functionalities are covered by tests. Comprehensive test coverage contributes to the overall reliability and quality of the Inventory class and the software system as a whole.

TEST CASE #3 (Keisi Breshanaj)

1. Introduction to Testing:

Software testing is a pivotal process integral to the software development lifecycle, encompassing the systematic evaluation of software to pinpoint defects or bugs. It serves as a critical safeguard, ensuring the reliability and correctness of software systems by subjecting them to various scenarios and inputs. The importance of testing cannot be overstated, as it significantly contributes to the overall quality of the software product. By identifying and rectifying defects early in the development process, testing minimizes the likelihood of issues affecting the software's performance or functionality, ultimately enhancing user satisfaction and trust.

2. Purpose of Testing:

In the context of the Cashier class, testing ensures that cashier-related functionalities, such as user credential generation and employee data management, operate seamlessly and adhere to specified requirements. By detecting defects early, testing mitigates the risk of issues proliferating throughout the software system, thereby reducing the cost and effort required for remediation.

3. Focus on Testing a Single Component:

The Cashier class serves an important function in the software system for managing cashier information, which emphasizes the significance of thorough testing. This part must perform accurately because it is in charge of important functions including processing employee data and creating user credentials. Developers can verify that the Cashier class behaves as planned and complies with business rules controlling cashier management by testing it in a variety of scenarios.

4. Preparing Test Cases:

Comprehensive test cases are essential for thorough testing of the Cashier class. These cases should encompass a spectrum of scenarios, ranging from normal inputs to edge cases and invalid inputs. For instance, test cases may include validating the generation of user credentials with valid input data, handling empty phone numbers or default salary values as edge cases, and verifying the class's behavior when confronted with invalid inputs, such as negative salary values. By preparing these test cases, developers can ensure exhaustive coverage of the Cashier class's functionalities and behaviors.

5. Choosing Testing Frameworks:

Choosing a good testing framework such as JUnit makes testing more organized and productive when it comes to Java programming. JUnit is a great option for verifying the functionality of the Cashier class because it offers powerful facilities for creating, running, and maintaining unit tests. In order to set up the JUnit testing environment, the framework must be included into the project's build settings. This will streamline the testing procedure and increase its overall effectiveness.

6. Writing Test Code:

Writing test code for the Cashier class entails creating test methods that exercise its various functionalities. These methods should be crafted to cover different scenarios and inputs, ensuring comprehensive testing coverage. Employing assertions within test methods enables developers to

validate expected outcomes against actual results, effectively verifying the correctness and reliability of the Cashier class's behavior.

```
import static org.junit.Assert.*;

import org.junit.Before;

import org.junit.Test;


public class CashierTest {


   private Cashier cashier;


   @Before

   public void setUp() {

      cashier = new Cashier("John", "Doe", "123456789");

   }


   @Test

   public void testGetId() {

      assertEquals(1, cashier.getId());

   }


   @Test

   public void testGetName() {

      assertEquals("John", cashier.getName());

   }


   @Test

   public void testGetSname() {

      assertEquals("Doe", cashier.getSname());

   }
```

```
@Test
public void testGetPhone() {
   assertEquals("123456789", cashier.getPhone());
}

@Test
public void testSetPhone() {
   cashier.setPhone("987654321");
   assertEquals("987654321", cashier.getPhone());
}

@Test
public void testGetSalary() {
   assertEquals(0.0, cashier.getSalary(), 0.01);
}

@Test
public void testSetSalary() {
   cashier.setSalary(1000.0);
   assertEquals(1000.0, cashier.getSalary(), 0.01);
}

@Test
public void testGenUser() {
   User user = cashier.genUser();
   assertEquals("jdoe", user.getUser());
   assertEquals("do02", user.getPass());
   assertEquals("Cashier", user.getStatus());
```

```
    }

    @Test
    public void testToString() {
        String expected = "Employee name: John\n Surname: Doe\n Phone: 123456789\n salary: 0.0";
        assertEquals(expected, cashier.toString());
    }
}
```

7. Running Tests:

Executing tests for the Cashier class and interpreting the results are critical steps in the testing process. By running tests, developers can identify passing, failing, or error scenarios, gaining insights into the class's behavior under different conditions. Interpreting test results involves analyzing the outcome of each test case, identifying discrepancies between expected and actual behavior, and taking corrective actions as necessary to rectify any issues.

testGetId: Checks if the getId method returns the unique identifier of the cashier.

testGetName: Verifies if the getName method retrieves the first name of the cashier.

testGetPhone: Ensures that the getPhone method retrieves the phone number of the cashier.

testGenUser: Validates if the genUser method generates user credentials based on the cashier's name and surname.

8. Test Coverage:

These test cases cover various functionalities of the Cashier class, including getters and setters for name, surname, phone, and salary, as well as the generation of user credentials and the toString() method. Each test method verifies that the actual output matches the expected output for the respective functionality. This not only helps identify areas of the code that require further testing but also enhances the overall reliability and quality of the software product, instilling confidence in its performance and functionality.

TEST CASE #4 (Loard Bejko)

1. Introduction to Testing:

Software testing is the systematic process of evaluating software to identify defects or bugs. In the context of the Product class, testing involves analyzing its functionality, attributes, and validation mechanisms to ensure that it behaves as expected and meets specified requirements. The importance of testing in software development for ensuring reliability and correctness cannot be overstated.

2. Purpose of Testing:

The goal of testing is to find flaws early in the development process so that they can be fixed quickly and don't affect other processes. Testing helps preserve the integrity of the system and improves its reliability by confirming that the Product class and its related features operate as intended. This ultimately helps to produce high-quality software solutions.

3. Focus on Testing a Single Component:

Testing the Product class is essential due to its central role in managing product data within the system. With its multifaceted responsibilities and complex attributes, including validation logic, defects within this class can have significant repercussions on inventory management, order processing, and customer experience. Thorough testing helps identify and rectify potential issues early, mitigating risks, improving software quality, and ensuring the reliability and stability of the system, ultimately enhancing overall performance and user satisfaction.

4. Preparing Test Cases:

Test cases for the Product class should cover a range of scenarios, including valid inputs, boundary cases, and invalid inputs. For instance, test cases may validate the setting and retrieval of product attributes, as well as the behavior of the isValid method for input validation.

5. Choosing Testing Frameworks:

To execute tests for the Product class, you would typically run the ProductTest class using a testing framework like JUnit. This framework automatically initializes the necessary objects and executes the test methods defined within the class.

6. Writing Test Code:

Test code for the Product class involves creating test methods to verify its functionalities. These methods should test various aspects of the class, such as attribute setting and retrieval, validation of input data, and the toString method for generating product information.

testIsValid_ValidData: Checks if the isValid method returns "1" for valid product data.

testIsValid_InvalidISBN: Tests if the isValid method returns the correct error message for an invalid ISBN.

testIsValid_NegativeSellingPrice: Ensures that the isValid method returns the correct error message for a negative selling price.

testIsValid_InvalidName: Validates if the isValid method returns the correct error message for an invalid product name.

testIsValid_NegativeStock: Verifies that the isValid method returns the correct error message for a negative stock quantity.

```
import static org.junit.Assert.*;

import org.junit.Test;


public class ProductTest {


   @Test

   public void testSetName() {

      Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", 10, "Supplier1", "2024-05-03", 25.0);

      product.setName("New Name");

      assertEquals("New Name", product.getName());

   }


   @Test

   public void testSetOrigPrice() {

      Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", 10, "Supplier1", "2024-05-03", 25.0);

      product.setOrigPrice(30.0);

      assertEquals(30.0, product.getOrigPrice(), 0.01);

   }


   @Test

   public void testSetSellingPrice() {

      Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", 10, "Supplier1", "2024-05-03", 25.0);
```

```
        product.setSellingPrice(30.0);
        assertEquals(30.0, product.getSellingPrice(), 0.01);
    }

    @Test
    public void testIsValid_ValidData() {
        Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", 10,
"Supplier1", "2024-05-03", 25.0);
        String result = product.isValid();
        assertEquals("1", result);
    }

    @Test
    public void testIsValid_InvalidISBN() {
        Product product = new Product("12345", "Test Product", 20.0, "Electronics", 10, "Supplier1", "2024-
05-03", 25.0);
        String result = product.isValid();
        assertEquals("ISBN must contain exactly 13 digits with no spaces/dashes.", result);
    }

    @Test
    public void testIsValid_NegativeSellingPrice() {
        Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", 10,
"Supplier1", "2024-05-03", -25.0);
        String result = product.isValid();
        assertEquals("Selling Price cannot be negative.", result);
    }

    @Test
    public void testIsValid_InvalidName() {
```

```
    Product product = new Product("1234567890123", "Test Product
12345678901234567890123456789012345678901234567890", 20.0, "Electronics", 10, "Supplier1", "2024-05-03", 25.0);

    String result = product.isValid();

    assertEquals("Title must contain 1 to 30 lower/upper case letters numbers spaces or underscore.",
result);

  }


  @Test

  public void testIsValid_NegativeStock() {

    Product product = new Product("1234567890123", "Test Product", 20.0, "Electronics", -10,
"Supplier1", "2024-05-03", 25.0);

    String result = product.isValid();

    assertEquals("Quantity cannot be negative.", result);

  }

}
```

7. Running Tests:

During the execution of tests, passing scenarios occur when the actual behavior matches the expected behavior defined within the test cases. For example, in the testSetName method, if the setName method successfully sets the name of the product and the assertion assertEquals("New Name", product.getName()) passes, the test case is marked as passing. Failing scenarios occur when the actual behavior deviates from the expected behavior defined within the test cases. This could happen due to issues such as incorrect implementation or unexpected behavior within the code being tested. For instance, if the testIsValid_InvalidISBN method expects the isValid method to return an error message for an invalid ISBN but instead receives a different message or passes unexpectedly, the test case is marked as failing. Error scenarios may arise due to exceptions or unforeseen circumstances encountered during test execution. For example, if a method call within a test case throws an unexpected exception, the test case would result in an error. It's crucial to investigate the cause of the error and address any underlying issues in the code to ensure accurate test results.

8. Test Coverage:

Achieving high test coverage is essential for thorough testing of the Product class. It ensures that all aspects of the class, including methods and edge cases, are correct and the ERP system will work without any defects. High test coverage contributes to the overall reliability and quality of the software product.