# SQA Assignment 1

Submitted By: `The Debugging Dead`

- **Md. Ehsan Khan** `011201122`
- **Shofi Rayhan Siyam** `011201117`
- **Muhammad Rahat Ahasan** `011201112`
- **Abdullah Tawhid** `011192035`

Section: **B**

Course: **CSE 4495**

Submitted To: **Md. Mohaiminul Islam**

Date: **1st Feb 2024**

## Abstract

This report presents an in-depth analysis of the application, focusing on the unit testing, code coverage, identified bugs, and corresponding recommended fixes. The unit tests have been executed to ensure the robustness of the system, revealing several critical bugs and areas where the code fails to handle certain edge cases or operational scenarios. The bugs were listed, and the source of each was traced to specific locations within the application. For each issue, a recommended fix has been provided, aiming to resolve the problem while improving the overall reliability and efficiency of the application. Additionally, the coverage report illustrates the areas of the code that are adequately tested, highlighting any gaps in testing that could benefit from further attention.

# Index



# Links

- Repository: Github
- Unit Test (Markdown): Unit Test Report
- List Of Bugs (Markdown): List Of Bugs
- Branch Coverage (Markdown): Branch Coverage

# Unit Testing Report

## Test `RecipeTest::testDefaultValues`

### Test ID: `1`

### Method

```
@Test
void testDefaultValues() {
    assertEquals("", recipe.getName());
    assertEquals(0, recipe.getPrice());
    assertEquals(0, recipe.getAmtCoffee());
    assertEquals(0, recipe.getAmtMilk());
    assertEquals(0, recipe.getAmtSugar());
    assertEquals(0, recipe.getAmtChocolate());
}
```

### Purpose

This test verifies that a newly created `Recipe` object initializes its attributes with default values. Specifically, it checks that the name is an empty string, and the price, coffee amount, milk amount, sugar amount, and chocolate amount are all set to 0.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetNameValid`

### Test ID: `2`

### Method

```
@Test
void testSetNameValid() {
    recipe.setName("Espresso");
    assertEquals("Espresso", recipe.getName(), "Name should be set to Espresso");
}
```

### Purpose

This test ensures that the `setName` method correctly updates the name of the recipe when a valid string is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetNameNull`

### Test ID: `3`

### Method

```
@Test
void testSetNameNull() {
    recipe.setName(null);
    assertEquals("", recipe.getName(), "Name should remain unchanged");
}
```

### Purpose

This test checks that the `setName` method handles a `null` input gracefully by retaining the default empty string value for the name.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceValid`

### Test ID: `4`

### Method

```
@Test
void testSetPriceValid() throws RecipeException {
    recipe.setPrice("100");
    assertEquals(100, recipe.getPrice(), "Price should be set to 100");
}
```

### Purpose

This test verifies that the `setPrice` method correctly parses and sets the price when a valid integer string is provided.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceNegative`

### Test ID: `5`

### Method

```
@Test
void testSetPriceNegative() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setPrice("-10"), "Negative Price Should Fail");
    assertEquals("Price must be a positive integer", exception.getMessage());
}
```

### Purpose

This test ensures that the `setPrice` method throws a `RecipeException` when a negative value is provided, as prices cannot be negative.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceInvalidFormat`

### Test ID: `6`

### Method

```
@Test
void testSetPriceInvalidFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setPrice("abc"), "Invalid Format Price Should Fail");
    assertEquals("Price must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setPrice` method throws a `RecipeException` when an invalid format (non-integer string) is provided.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceZero`

### Test ID: `7`

### Method

```
@Test
void testSetPriceZero() throws RecipeException {
    recipe.setPrice("0");
    assertEquals(0, recipe.getPrice(), "Price should be set to 0");
}
```

### Purpose

This test verifies that the `setPrice` method correctly handles a zero value, which is considered valid.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceLargeValue`

### Test ID: `8`

### Method

```
@Test
void testSetPriceLargeValue() throws RecipeException {
    recipe.setPrice("1000000");
    assertEquals(1000000, recipe.getPrice(), "Price should be set to 1,000,000");
}
```

### Purpose

This test ensures that the `setPrice` method can handle large integer values without issues.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceWhitespace`

### Test ID: `9`

### Method

```
@Test
void testSetPriceWhitespace() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setPrice("   "), "Whitespace Price Should Fail");
    assertEquals("Price must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setPrice` method throws a `RecipeException` when the input consists solely of whitespace.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetPriceFloatInput`

### Test ID: `10`

### Method

```
@Test
void testSetPriceFloatInput() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setPrice("10.5"), "Float Input Price Should Fail");
}
```

### Purpose

This test ensures that the `setPrice` method throws a `RecipeException` when a floating-point number is provided, as only integers are allowed.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetAmtCoffeeValid`

### Test ID: `11`

### Method

```
@Test
void testSetAmtCoffeeValid() throws RecipeException {
    recipe.setAmtCoffee("5");
    assertEquals(5, recipe.getAmtCoffee(), "Coffee Amount should be set to 5");
}
```

### Purpose

This test verifies that the `setAmtCoffee` method correctly sets the coffee amount when a valid integer string is provided.

### Execution Report: `PASSED`

## Test `RecipeTest::testSetAmtCoffeeZero`

### Test ID: `12`

### Method

```
@Test
void testSetAmtCoffeeZero() throws RecipeException {
    recipe.setAmtCoffee("0");
    assertEquals(0, recipe.getAmtCoffee(), "Coffee Amount should be set to 0");
}
```

### Purpose

This test ensures that the `setAmtCoffee` method handles a zero value correctly, as zero is a valid input.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtCoffeeNonIntegerFormat`

### Test ID: 13

### Method

```
@Test
void testSetAmtCoffeeNonIntegerFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtCoffee("abc"), "Non-Integer Format Coffee Amount Sh
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setAmtCoffee` method throws a `RecipeException` when a non-integer string is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtCoffeeNull`

### Test ID: 14

### Method

```
@Test
void testSetAmtCoffeeNull() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtCoffee(null), "Null Coffee Amount Should Fail");
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
}
```

### Purpose

This test ensures that the `setAmtCoffee` method throws a `RecipeException` when a `null` value is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtCoffeeNegative`

### Test ID: 15

### Method

```
@Test
void testSetAmtCoffeeNegative() {
```

```
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtCoffee("-1"), "Negative Coffee Amount Should Fail")
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
}
```

### Purpose

This test verifies that the `setAmtCoffee` method throws a `RecipeException` when a negative value is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtCoffeeInvalidFormat`

### Test ID: **16**

### Method

```
@Test
void testSetAmtCoffeeInvalidFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtCoffee("xyz"), "Invalid Format Coffee Amount Should
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setAmtCoffee` method throws a `RecipeException` when an invalid format (non-integer string) is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtMilkValid`

### Test ID: **17**

### Method

```
@Test
void testSetAmtMilkValid() throws RecipeException {
    recipe.setAmtMilk("3");
    assertEquals(3, recipe.getAmtMilk(), "Milk Amount should be set to 3");
}
```

### Purpose

This test verifies that the `setAmtMilk` method correctly sets the milk amount when a valid integer string is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtMilkZero`

### Test ID: **18**

### Method

```
@Test
void testSetAmtMilkZero() throws RecipeException {
    recipe.setAmtMilk("0");
```

```
    assertEquals(0, recipe.getAmtMilk(), "Milk Amount should be set to 0");
}
```

### Purpose

This test ensures that the `setAmtMilk` method handles a zero value correctly, as zero is a valid input.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtMilkNegative`

### Test ID: 19

### Method

```
@Test
void testSetAmtMilkNegative() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtMilk("-2"), "Negative Milk Amount Should Fail");
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setAmtMilk` method throws a `RecipeException` when a negative value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtMilkNull`

### Test ID: 20

### Method

```
@Test
void testSetAmtMilkNull() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtMilk(null), "Null Milk Amount Should Fail");
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
}
```

### Purpose

This test ensures that the `setAmtMilk` method throws a `RecipeException` when a `null` value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtMilkInvalidFormat`

### Test ID: 21

### Method

```
@Test
void testSetAmtMilkInvalidFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtMilk("milk"), "Invalid Format Milk Amount Should Fa
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
}
```

### Purpose

This test verifies that the `setAmtMilk` method throws a `RecipeException` when an invalid format (non-integer string) is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtSugarValid`

### Test ID: **22**

### Method

```
@Test
void testSetAmtSugarValid() throws RecipeException {
    recipe.setAmtSugar("4");
    assertEquals(4, recipe.getAmtSugar(), "Sugar Amount should be set to 4");
}
```

### Purpose

This test ensures that the `setAmtSugar` method correctly sets the sugar amount when a valid integer string is provided.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtSugarZero`

### Test ID: **23**

### Method

```
@Test
void testSetAmtSugarZero() throws RecipeException {
    recipe.setAmtSugar("0");
    assertEquals(0, recipe.getAmtSugar(), "Sugar Amount should be set to 0");
}
```

### Purpose

This test verifies that the `setAmtSugar` method handles a zero value correctly, as zero is a valid input.

### Execution Report: **PASSED**

## Test `RecipeTest::testSetAmtSugarNegative`

### Test ID: **24**

### Method

```
@Test
void testSetAmtSugarNegative() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtSugar("-3"), "Negative Sugar Amount Should Fail");
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setAmtSugar` method throws a `RecipeException` when a negative value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtSugarNull`

### Test ID: 25

### Method

```
@Test
void testSetAmtSugarNull() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtSugar(null), "Null Sugar Amount Should Fail");
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
}
```

### Purpose

This test ensures that the `setAmtSugar` method throws a `RecipeException` when a `null` value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtSugarInvalidFormat`

### Test ID: 26

### Method

```
@Test
void testSetAmtSugarInvalidFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtSugar("sweet"), "Invalid Format Sugar Amount Should
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
}
```

### Purpose

This test verifies that the `setAmtSugar` method throws a `RecipeException` when an invalid format (non-integer string) is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtChocolateValid`

### Test ID: 27

### Method

```
@Test
void testSetAmtChocolateValid() throws RecipeException {
    recipe.setAmtChocolate("6");
    assertEquals(6, recipe.getAmtChocolate(), "Chocolate Amount should be set to 6");
}
```

### Purpose

This test ensures that the `setAmtChocolate` method correctly sets the chocolate amount when a valid integer string is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtChocolateZero`

### Test ID: 28

### Method

```
@Test
void testSetAmtChocolateZero() throws RecipeException {
    recipe.setAmtChocolate("0");
    assertEquals(0, recipe.getAmtChocolate(), "Chocolate Amount should be set to 0");
}
```

### Purpose

This test verifies that the `setAmtChocolate` method handles a zero value correctly, as zero is a valid input.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtChocolateNull`

### Test ID: 29

### Method

```
@Test
void testSetAmtChocolateNull() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtChocolate(null), "Null Chocolate Amount Should Fail
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
}
```

### Purpose

This test checks that the `setAmtChocolate` method throws a `RecipeException` when a `null` value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtChocolateNegative`

### Test ID: 30

### Method

```
@Test
void testSetAmtChocolateNegative() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtChocolate("-5"), "Negative Chocolate Amount Should
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
}
```

### Purpose

This test ensures that the `setAmtChocolate` method throws a `RecipeException` when a negative value is provided.

### Execution Report: PASSED

## Test `RecipeTest::testSetAmtChocolateInvalidFormat`

### Test ID: `31`

### Method

```
@Test
void testSetAmtChocolateInvalidFormat() {
    RecipeException exception = assertThrows(RecipeException.class, () -> recipe.setAmtChocolate("dark"), "Invalid Format Chocolate Amount
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
}
```

### Purpose

This test verifies that the `setAmtChocolate` method throws a `RecipeException` when an invalid format (non-integer string) is provided.

### Execution Report: `PASSED`

## Test `RecipeTest::testToString`

### Test ID: `32`

### Method

```
@Test
void testToString() {
    recipe.setName("Cappuccino");
    assertEquals("Cappuccino", recipe.toString());
}
```

### Purpose

This test checks that the `toString` method returns the correct string representation of the recipe, which is the recipe's name.

### Execution Report: `PASSED`

## Test `RecipeTest::testEqualsNullObject`

### Test ID: `33`

### Method

```
@Test
void testEqualsNullObject() {
    assertNotEquals(null, recipe);
}
```

### Purpose

This test ensures that the `equals` method returns `false` when comparing a `Recipe` object to `null`.

### Execution Report: `PASSED`

## Test `RecipeTest::testEqualsDifferentClass`

### Test ID: 34

### Method

```
@Test
void testEqualsDifferentClass() {
    assertNotEquals(new Object(), recipe);
}
```

### Purpose

This test verifies that the `equals` method returns `false` when comparing a `Recipe` object to an object of a different class.

### Execution Report: PASSED

## Test `RecipeTest::testEqualsDifferentName`

### Test ID: 35

### Method

```
@Test
void testEqualsDifferentName() {
    Recipe anotherRecipe = new Recipe();
    anotherRecipe.setName("Latte");
    recipe.setName("Mocha");
    assertNotEquals(recipe, anotherRecipe);
}
```

### Purpose

This test checks that the `equals` method returns `false` when comparing two `Recipe` objects with different names.

### Execution Report: PASSED

## Test `RecipeTest::testEqualsSameName`

### Test ID: 36

### Method

```
@Test
void testEqualsSameName() {
    Recipe anotherRecipe = new Recipe();
    anotherRecipe.setName("Americano");
    recipe.setName("Americano");
    assertEquals(recipe, anotherRecipe);
}
```

### Purpose

This test verifies that the `equals` method returns `true` when comparing two `Recipe` objects with the same name.

### Execution Report: PASSED

## Test `RecipeTest::testHashCodeSame`

### Test ID: 37

### Method

```
@Test
void testHashCodeSame() {
    Recipe anotherRecipe = new Recipe();
    anotherRecipe.setName("Espresso");
    recipe.setName("Espresso");
    assertEquals(recipe.hashCode(), anotherRecipe.hashCode());
}
```

### Purpose

This test ensures that the `hashCode` method returns the same value for two `Recipe` objects with the same name.

### Execution Report: PASSED

## Test `RecipeTest::testHashCodeDifferent`

### Test ID: 38

### Method

```
@Test
void testHashCodeDifferent() {
    Recipe anotherRecipe = new Recipe();
    anotherRecipe.setName("Mocha");
    recipe.setName("Latte");
    assertNotEquals(recipe.hashCode(), anotherRecipe.hashCode());
}
```

### Purpose

This test checks that the `hashCode` method returns different values for two `Recipe` objects with different names.

### Execution Report: PASSED

## Test `InventoryTest::testDefaultValues`

### Test ID: 39

### Method

```
@Test
public void testDefaultValues() {
    assertEquals(15, inventory.getCoffee());
    assertEquals(15, inventory.getMilk());
    assertEquals(15, inventory.getSugar());
    assertEquals(15, inventory.getChocolate());
}
```

### Purpose

This test verifies that a newly created `Inventory` object initializes its attributes with default values. Specifically, it checks that the coffee, milk, sugar, and chocolate amounts are all set to 15.

### Execution Report: PASSED

## Test `InventoryTest::testSetCoffeeValid`

### Test ID: `40`

### Method

```
@Test
public void testSetCoffeeValid() {
    inventory.setCoffee(10);
    assertEquals(10, inventory.getCoffee(), "Coffee should be set to 10");
}
```

### Purpose

This test ensures that the `setCoffee` method correctly updates the coffee amount when a valid positive integer is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testSetCoffeeNegative`

### Test ID: `41`

### Method

```
@Test
public void testSetCoffeeNegative() {
    inventory.setCoffee(-5);
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when set to a negative value");
}
```

### Purpose

This test checks that the `setCoffee` method ignores negative values and retains the default coffee amount.

### Execution Report: `PASSED`

## Test `InventoryTest::testSetCoffeeZero`

### Test ID: `42`

### Method

```
@Test
public void testSetCoffeeZero() {
    inventory.setCoffee(0);
    assertEquals(0, inventory.getCoffee(), "Coffee should be set to 0");
}
```

### Purpose

This test verifies that the `setCoffee` method correctly handles a zero value.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddCoffeeValid`

### Test ID: 43

### Method

```
@Test
public void testAddCoffeeValid() throws InventoryException {
    inventory.addCoffee("5");
    assertEquals(20, inventory.getCoffee(), "Coffee should increase by 5");
}
```

### Purpose

This test ensures that the `addCoffee` method correctly increases the coffee amount when a valid integer string is provided.

### Execution Report: PASSED

## Test `InventoryTest::testAddCoffeeZero`

### Test ID: 44

### Method

```
@Test
public void testAddCoffeeZero() throws InventoryException {
    inventory.addCoffee("0");
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when adding 0");
}
```

### Purpose

This test checks that the `addCoffee` method does not change the coffee amount when a zero value is provided.

### Execution Report: PASSED

## Test `InventoryTest::testAddCoffeeNegativeValue`

### Test ID: 45

### Method

```
@Test
public void testAddCoffeeNegativeValue() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addCoffee("-10"), "Adding negative coffee should fail");
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when adding a negative value");
}
```

### Purpose

This test verifies that the `addCoffee` method throws an `InventoryException` when a negative value is provided.

### Execution Report: PASSED

## Test `InventoryTest::testAddCoffeeInvalidFormat`

### Test ID: 46

### Method

```
@Test
public void testAddCoffeeInvalidFormat() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addCoffee("abc"), "Invalid Format Coffee Should Fail");
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when adding an invalid string");
}
```

### Purpose

This test ensures that the `addCoffee` method throws an `InventoryException` when an invalid format (non-integer string) is provided.

### Execution Report: PASSED

## Test `InventoryTest::testAddCoffeeDecimalValue`

### Test ID: 47

### Method

```
@Test
public void testAddCoffeeDecimalValue() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addCoffee("10.5"), "Adding decimal coffee should fail");
    assertEquals("Units of coffee must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when adding a decimal value");
}
```

### Purpose

This test checks that the `addCoffee` method throws an `InventoryException` when a decimal value is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetMilkValid`

### Test ID: 48

### Method

```
@Test
public void testSetMilkValid() {
    inventory.setMilk(10);
    assertEquals(10, inventory.getMilk(), "Milk should be set to 10");
}
```

### Purpose

This test ensures that the `setMilk` method correctly updates the milk amount when a valid positive integer is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetMilkNegative`

### Test ID: 49

### Method

```
@Test
public void testSetMilkNegative() {
    inventory.setMilk(-5);
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when set to a negative value");
}
```

### Purpose

This test verifies that the `setMilk` method ignores negative values and retains the default milk amount.

### Execution Report: `PASSED`

## Test `InventoryTest::testSetMilkZero`

### Test ID: `50`

### Method

```
@Test
public void testSetMilkZero() {
    inventory.setMilk(0);
    assertEquals(0, inventory.getMilk(), "Milk should be set to 0");
}
```

### Purpose

This test checks that the `setMilk` method correctly handles a zero value.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddMilkValid`

### Test ID: `51`

### Method

```
@Test
public void testAddMilkValid() throws InventoryException {
    inventory.addMilk("5");
    assertEquals(20, inventory.getMilk(), "Milk should increase by 5");
}
```

### Purpose

This test ensures that the `addMilk` method correctly increases the milk amount when a valid integer string is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddMilkZero`

### Test ID: `52`

### Method

```java
@Test
public void testAddMilkZero() throws InventoryException {
    inventory.addMilk("0");
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when adding 0");
}
```

### Purpose

This test verifies that the `addMilk` method does not change the milk amount when a zero value is provided.

### Execution Report: **PASSED**

## Test `InventoryTest::testAddMilkNegative`

### Test ID: **53**

### Method

```java
@Test
public void testAddMilkNegative() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addMilk("-5"), "Negative Milk Should Fail");
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when adding a negative value");
}
```

### Purpose

This test checks that the `addMilk` method throws an `InventoryException` when a negative value is provided.

### Execution Report: **PASSED**

## Test `InventoryTest::testAddMilkInvalidFormat`

### Test ID: **54**

### Method

```java
@Test
public void testAddMilkInvalidFormat() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addMilk("abc"), "Invalid Format Milk Should Fail");
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when adding an invalid string");
}
```

### Purpose

This test ensures that the `addMilk` method throws an `InventoryException` when an invalid format (non-integer string) is provided.

### Execution Report: **PASSED**

## Test `InventoryTest::testAddMilkDecimalValue`

### Test ID: **55**

### Method

```java
@Test
public void testAddMilkDecimalValue() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addMilk("5.5"), "Decimal Milk Should Fail");
    assertEquals("Units of milk must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when adding a decimal value");
}
```

### Purpose

This test verifies that the `addMilk` method throws an `InventoryException` when a decimal value is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetSugarValid`

### Test ID: 56

### Method

```java
@Test
public void testSetSugarValid() {
    inventory.setSugar(10);
    assertEquals(10, inventory.getSugar(), "Sugar should be set to 10");
}
```

### Purpose

This test ensures that the `setSugar` method correctly updates the sugar amount when a valid positive integer is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetSugarNegative`

### Test ID: 57

### Method

```java
@Test
public void testSetSugarNegative() {
    inventory.setSugar(-5);
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when set to a negative value");
}
```

### Purpose

This test checks that the `setSugar` method ignores negative values and retains the default sugar amount.

### Execution Report: PASSED

## Test `InventoryTest::testSetSugarZero`

### Test ID: 58

### Method

```java
@Test
public void testSetSugarZero() {
```

```
    inventory.setSugar(0);
    assertEquals(0, inventory.getSugar(), "Sugar should be set to 0");
}
```

### Purpose

This test verifies that the `setSugar` method correctly handles a zero value.

### Execution Report: **PASSED**

## Test `InventoryTest::testAddSugarValid`

### Test ID: **59**

### Method

```
@Test
public void testAddSugarValid() {
    try {
        inventory.addSugar("5");
    } catch (InventoryException e) {
        fail("Adding sugar should not throw an exception");
    }
    assertEquals(20, inventory.getSugar(), "Sugar should increase by 5");
}
```

### Purpose

This test ensures that the `addSugar` method correctly increases the sugar amount when a valid integer string is provided.

### Execution Report: **FAILED**

```
org.opentest4j.AssertionFailedError: Adding sugar should not throw an exception
```

## Test `InventoryTest::testAddSugarZero`

### Test ID: **60**

### Method

```
@Test
public void testAddSugarZero() {
    try {
        inventory.addSugar("0");
    } catch (InventoryException e) {
        fail("Adding sugar should not throw an exception");
    }
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when adding 0");
}
```

### Purpose

This test checks that the `addSugar` method does not change the sugar amount when a zero value is provided.

### Execution Report: **PASSED**

## Test `InventoryTest::testAddSugarNegativeValue`

### Test ID: 61

### Method

```
@Test
public void testAddSugarNegativeValue() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addSugar("-10"), "Adding negative sugar should fail");
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when adding a negative value");
}
```

### Purpose

This test verifies that the `addSugar` method throws an `InventoryException` when a negative value is provided.

### Execution Report: FAILED

```
org.opentest4j.AssertionFailedError: Adding negative sugar should fail ==>
Expected coffee.exceptions.InventoryException to be thrown, but nothing was thrown.
```

## Test `InventoryTest::testAddSugarInvalidFormat`

### Test ID: 62

### Method

```
@Test
public void testAddSugarInvalidFormat() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addSugar("abc"), "Invalid Format Sugar Should Fail");
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when adding an invalid string");
}
```

### Purpose

This test ensures that the `addSugar` method throws an `InventoryException` when an invalid format (non-integer string) is provided.

### Execution Report: PASSED

## Test `InventoryTest::testAddSugarDecimalValue`

### Test ID: 63

### Method

```
@Test
public void testAddSugarDecimalValue() {
    Exception exception = assertThrows(InventoryException.class, () -> inventory.addSugar("10.5"), "Adding decimal sugar should fail");
    assertEquals("Units of sugar must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when adding a decimal value");
}
```

### Purpose

This test checks that the `addSugar` method throws an `InventoryException` when a decimal value is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetChocolateValid`

### Test ID: 64

### Method

```
@Test
public void testSetChocolateValid() {
    inventory.setChocolate(10);
    assertEquals(10, inventory.getChocolate(), "Chocolate should be set to 10");
}
```

### Purpose

This test ensures that the `setChocolate` method correctly updates the chocolate amount when a valid positive integer is provided.

### Execution Report: PASSED

## Test `InventoryTest::testSetChocolateNegative`

### Test ID: 65

### Method

```
@Test
public void testSetChocolateNegative() {
    inventory.setChocolate(-5);
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when set to a negative value");
}
```

### Purpose

This test verifies that the `setChocolate` method ignores negative values and retains the default chocolate amount.

### Execution Report: PASSED

## Test `InventoryTest::testSetChocolateZero`

### Test ID: 66

### Method

```
@Test
public void testSetChocolateZero() {
    inventory.setChocolate(0);
    assertEquals(0, inventory.getChocolate(), "Chocolate should be set to 0");
}
```

### Purpose

This test checks that the `setChocolate` method correctly handles a zero value.

### Execution Report: PASSED

## Test `InventoryTest::testAddChocolateValid`

### Test ID: `67`

### Method

```
@Test
public void testAddChocolateValid() throws InventoryException {
    inventory.addChocolate("5");
    assertEquals(20, inventory.getChocolate(), "Chocolate should increase by 5");
}
```

### Purpose

This test ensures that the `addChocolate` method correctly increases the chocolate amount when a valid integer string is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddChocolateZero`

### Test ID: `68`

### Method

```
@Test
public void testAddChocolateZero() throws InventoryException {
    inventory.addChocolate("0");
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when adding 0");
}
```

### Purpose

This test verifies that the `addChocolate` method does not change the chocolate amount when a zero value is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddChocolateNegativeValue`

### Test ID: `69`

### Method

```
@Test
public void testAddChocolateNegativeValue() {
    Exception exception = assertThrows(InventoryException.class,
        () -> inventory.addChocolate("-5"),
        "Adding negative chocolate should fail");
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when adding a negative value");
}
```

### Purpose

This test checks that the `addChocolate` method throws an `InventoryException` when a negative value is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddChocolateInvalidFormat`

### Test ID: `70`

### Method

```
@Test
public void testAddChocolateInvalidFormat() {
    Exception exception = assertThrows(InventoryException.class,
        () -> inventory.addChocolate("abc"),
        "Adding invalid format for chocolate should fail");
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when adding an invalid string");
}
```

### Purpose

This test ensures that the `addChocolate` method throws an `InventoryException` when an invalid format (non-integer string) is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testAddChocolateDecimalValue`

### Test ID: `71`

### Method

```
@Test
public void testAddChocolateDecimalValue() {
    Exception exception = assertThrows(InventoryException.class,
        () -> inventory.addChocolate("10.5"),
        "Adding decimal chocolate should fail");
    assertEquals("Units of chocolate must be a positive integer", exception.getMessage());
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when adding a decimal value");
}
```

### Purpose

This test verifies that the `addChocolate` method throws an `InventoryException` when a decimal value is provided.

### Execution Report: `PASSED`

## Test `InventoryTest::testEnoughIngredientsSufficient`

### Test ID: `72`

### Method

```
@Test
public void testEnoughIngredientsSufficient() {
    try {
        recipe.setAmtCoffee("5");
        recipe.setAmtMilk("5");
        recipe.setAmtSugar("5");
        recipe.setAmtChocolate("5");
    } catch (RecipeException e) {
        fail("Adding ingredients should not throw an exception");
    }
    assertTrue(inventory.enoughIngredients(recipe), "Should return true when there are sufficient ingredients");
}
```

### Purpose

This test checks that the `enoughIngredients` method returns `true` when the inventory has sufficient ingredients for the recipe.

### Execution Report: PASSED

## Test `InventoryTest::testEnoughIngredientsInsufficientCoffee`

### Test ID: 73

### Method

```
@Test
public void testEnoughIngredientsInsufficientCoffee() {
    try {
        inventory.setCoffee(0);
        recipe.setAmtCoffee("5");
    } catch (RecipeException e) {
        fail("Adding ingredients should not throw an exception");
    }
    assertFalse(inventory.enoughIngredients(recipe), "Should return false when coffee is insufficient");
}
```

### Purpose

This test verifies that the `enoughIngredients` method returns `false` when the coffee amount is insufficient for the recipe.

### Execution Report: PASSED

## Test `InventoryTest::testEnoughIngredientsInsufficientMilk`

### Test ID: 74

### Method

```
@Test
public void testEnoughIngredientsInsufficientMilk() {
    try {
        inventory.setMilk(0);
        recipe.setAmtMilk("5");
    } catch (RecipeException e) {
        fail("Adding ingredients should not throw an exception");
    }
    assertFalse(inventory.enoughIngredients(recipe), "Should return false when milk is insufficient");
}
```

### Purpose

This test ensures that the `enoughIngredients` method returns `false` when the milk amount is insufficient for the recipe.

### Execution Report: PASSED

## Test `InventoryTest::testEnoughIngredientsInsufficientSugar`

### Test ID: 75

### Method

```
@Test
public void testEnoughIngredientsInsufficientSugar() {
    try {
```

```
        inventory.setSugar(0);
        recipe.setAmtSugar("5");
    } catch (RecipeException e) {
        fail("Adding ingredients should not throw an exception");
    }
    assertFalse(inventory.enoughIngredients(recipe), "Should return false when sugar is insufficient");
}
```

### Purpose

This test checks that the `enoughIngredients` method returns `false` when the sugar amount is insufficient for the recipe.

### Execution Report: **PASSED**

## Test `InventoryTest::testEnoughIngredientsInsufficientChocolate`

### Test ID: **76**

### Method

```
@Test
public void testEnoughIngredientsInsufficientChocolate() {
    try {
        inventory.setChocolate(0);
        recipe.setAmtChocolate("5");
    } catch (RecipeException e) {
        fail("Adding ingredients should not throw an exception");
    }
    assertFalse(inventory.enoughIngredients(recipe), "Should return false when chocolate is insufficient");
}
```

### Purpose

This test verifies that the `enoughIngredients` method returns `false` when the chocolate amount is insufficient for the recipe.

### Execution Report: **PASSED**

## Test `InventoryTest::testUseIngredientsSufficient`

### Test ID: **77**

### Method

```
@Test
public void testUseIngredientsSufficient() {
    try {
        recipe.setAmtCoffee("5");
        recipe.setAmtMilk("5");
        recipe.setAmtSugar("5");
        recipe.setAmtChocolate("5");
    } catch (RecipeException e) {
        fail("Setting recipe amounts should not throw an exception");
    }

    assertAll("Inventory should have enough ingredients to make the recipe",
        () -> assertTrue(inventory.useIngredients(recipe), "useIngredients should return true when ingredients are sufficient"),
        () -> assertEquals(10, inventory.getCoffee(), "Coffee should decrease by 5"),
        () -> assertEquals(10, inventory.getMilk(), "Milk should decrease by 5"),
        () -> assertEquals(10, inventory.getSugar(), "Sugar should decrease by 5"),
        () -> assertEquals(10, inventory.getChocolate(), "Chocolate should decrease by 5")
    );
}
```

### Purpose

This test ensures that the `useIngredients` method correctly reduces the inventory amounts when there are sufficient ingredients for the recipe.

### Execution Report: **FAILED**

```
org.opentest4j.AssertionFailedError: Coffee should decrease by 5 ==>
Expected :10
Actual   :20
```

## Test `InventoryTest::testUseIngredientsInsufficient`

### Test ID: **78**

### Method

```java
@Test
public void testUseIngredientsInsufficient() {
    try {
        inventory.setMilk(2);
        recipe.setAmtCoffee("5");
        recipe.setAmtMilk("5");
    } catch (RecipeException e) {
        fail("Setting recipe amounts should not throw an exception");
    }
    assertFalse(inventory.useIngredients(recipe), "useIngredients should return false when ingredients are insufficient");
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when ingredients are insufficient");
    assertEquals(2, inventory.getMilk(), "Milk should remain unchanged when ingredients are insufficient");
}
```

### Purpose

This test checks that the `useIngredients` method returns `false` and does not modify the inventory when there are insufficient ingredients for the recipe.

### Execution Report: **PASSED**

## Test `InventoryTest::testUseIngredientsZeroAmounts`

### Test ID: **79**

### Method

```java
@Test
public void testUseIngredientsZeroAmounts() {
    try {
        recipe.setAmtCoffee("0");
        recipe.setAmtMilk("0");
        recipe.setAmtSugar("0");
        recipe.setAmtChocolate("0");
    } catch (RecipeException e) {
        fail("Setting recipe amounts should not throw an exception");
    }
    assertTrue(inventory.useIngredients(recipe), "useIngredients should return true when the recipe requires zero amounts");
    assertEquals(15, inventory.getCoffee(), "Coffee should remain unchanged when the recipe requires zero amounts");
    assertEquals(15, inventory.getMilk(), "Milk should remain unchanged when the recipe requires zero amounts");
    assertEquals(15, inventory.getSugar(), "Sugar should remain unchanged when the recipe requires zero amounts");
    assertEquals(15, inventory.getChocolate(), "Chocolate should remain unchanged when the recipe requires zero amounts");
}
```

### Purpose

This test verifies that the `useIngredients` method returns `true` and does not modify the inventory when the recipe requires zero amounts of all ingredients.

### Execution Report: `PASSED`

## Test `InventoryTest::testToString`

### Test ID: `80`

### Method

```
@Test
public void testToString() {
    String expected = "Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n";
    assertEquals(expected, inventory.toString());
}
```

### Purpose

This test ensures that the `toString` method returns the correct string representation of the inventory.

### Execution Report: `PASSED`

## Test `RecipeBookTest::testDefaultValues`

### Test ID: `81`

### Method

```
@Test
public void testDefaultValues() {
    Recipe[] recipes = recipeBook.getRecipes();
    assertNotNull(recipes, "Recipe array should not be null");
    assertEquals(4, recipes.length, "Initial Recipe array should have length 4");
    for (Recipe recipe : recipes) {
        assertNull(recipe, "Initial Recipe array should be empty");
    }
}
```

### Purpose

This test verifies that a newly created `RecipeBook` object initializes its recipe array with a length of 4 and all elements are set to `null`.

### Execution Report: `PASSED`

## Test `RecipeBookTest::testAddRecipeSuccess`

### Test ID: `82`

### Method

```
@Test
public void testAddRecipeSuccess() {
    assertTrue(recipeBook.addRecipe(recipe1));
    assertTrue(recipeBook.addRecipe(recipe2));
    assertTrue(recipeBook.addRecipe(recipe3));
    assertTrue(recipeBook.addRecipe(recipe4));
}
```

### Purpose

This test ensures that the `addRecipe` method successfully adds up to 4 unique recipes to the recipe book.

### Execution Report: **PASSED**

## Test `RecipeBookTest::testAddRecipeDuplicate`

### Test ID: **83**

### Method

```
@Test
public void testAddRecipeDuplicate() {
    recipeBook.addRecipe(recipe1);
    assertFalse(recipeBook.addRecipe(recipe1), "Duplicate recipe should not be added");
}
```

### Purpose

This test checks that the `addRecipe` method returns `false` when attempting to add a duplicate recipe.

### Execution Report: **PASSED**

## Test `RecipeBookTest::testAddRecipeDuplicateWithDifferentObject`

### Test ID: **84**

### Method

```
@Test
public void testAddRecipeDuplicateWithDifferentObject() {
    Recipe duplicateRecipe = new Recipe();
    duplicateRecipe.setName("Recipe1");
    recipeBook.addRecipe(recipe1);
    assertFalse(recipeBook.addRecipe(duplicateRecipe), "Recipes with identical contents should be treated as duplicates");
}
```

### Purpose

This test verifies that the `addRecipe` method treats recipes with identical names as duplicates, even if they are different objects.

### Execution Report: **PASSED**

## Test `RecipeBookTest::testAddRecipeFull`

### Test ID: **85**

### Method

```
@Test
public void testAddRecipeFull() {
    recipeBook.addRecipe(recipe1);
    recipeBook.addRecipe(recipe2);
    recipeBook.addRecipe(recipe3);
    recipeBook.addRecipe(recipe4);
    assertFalse(recipeBook.addRecipe(recipe5), "Recipe book is full, so recipe5 should not be added");
}
```

### Purpose

This test ensures that the `addRecipe` method returns `false` when attempting to add a recipe to a full recipe book.

### Execution Report: **PASSED**

## Test `RecipeBookTest::testAddRecipeNull`

### Test ID: **86**

### Method

```
@Test
public void testAddRecipeNull() {
    try {
        assertFalse(recipeBook.addRecipe(null), "Adding a null recipe should return false");
    } catch (NullPointerException e) {
        fail("throwing NullPointerException instead of returning false");
    }
}
```

### Purpose

This test checks that the `addRecipe` method returns `false` when attempting to add a `null` recipe, without throwing an exception.

### Execution Report: **FAILED**

```
org.opentest4j.AssertionFailedError: throwing NullPointerException instead of returning false
```

## Test `RecipeBookTest::testAddRecipeDoesNotOverwrite`

### Test ID: **87**

### Method

```
@Test
public void testAddRecipeDoesNotOverwrite() {
    recipeBook.addRecipe(recipe1);
    recipeBook.addRecipe(recipe2);
    Recipe[] recipes = recipeBook.getRecipes();
    assertEquals(recipe1, recipes[0], "Recipe1 should remain at index 0");
    assertEquals(recipe2, recipes[1], "Recipe2 should remain at index 1");
    assertNull(recipes[2], "Index 2 should be null");
}
```

### Purpose

This test verifies that adding recipes does not overwrite existing recipes in the recipe book.

### Execution Report: **PASSED**

## Test `RecipeBookTest::testDeleteRecipeSuccess`

### Test ID: **88**

### Method

```java
@Test
public void testDeleteRecipeSuccess() {
    recipeBook.addRecipe(recipe1);
    assertEquals(recipe1.getName(), recipeBook.deleteRecipe(0));
    assertNull(recipeBook.getRecipes()[0], "Recipe1 should be deleted");
}
```

### Purpose

This test ensures that the `deleteRecipe` method successfully deletes a recipe at a valid index and returns the name of the deleted recipe.

### Execution Report: `FAILED`

```
org.opentest4j.AssertionFailedError: Recipe1 should be deleted ==>
Expected :null
Actual   :
```

## Test `RecipeBookTest::testDeleteRecipeInvalidIndexNegative`

### Test ID: `89`

### Method

```java
@Test
public void testDeleteRecipeInvalidIndexNegative() {
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        recipeBook.deleteRecipe(-1);
    }, "Negative index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test checks that the `deleteRecipe` method throws an `ArrayIndexOutOfBoundsException` when a negative index is provided.

### Execution Report: `PASSED`

## Test `RecipeBookTest::testDeleteRecipeInvalidIndexOutOfBounds`

### Test ID: `90`

### Method

```java
@Test
public void testDeleteRecipeInvalidIndexOutOfBounds() {
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        recipeBook.deleteRecipe(4);
    }, "Out of bounds index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test verifies that the `deleteRecipe` method throws an `ArrayIndexOutOfBoundsException` when an out-of-bounds index is provided.

### Execution Report: `PASSED`

## Test `RecipeBookTest::testDeleteRecipeDoesNotExist`

### Test ID: 91

### Method

```
@Test
public void testDeleteRecipeDoesNotExist() {
    assertNull(recipeBook.deleteRecipe(0), "No recipe at index 0, so should return null");
}
```

### Purpose

This test ensures that the `deleteRecipe` method returns `null` when attempting to delete a recipe from an empty index.

### Execution Report: PASSED

## Test `RecipeBookTest::testEditRecipeSuccess`

### Test ID: 92

### Method

```
@Test
public void testEditRecipeSuccess() {
    recipeBook.addRecipe(recipe1);
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertEquals("Recipe1", recipeBook.editRecipe(0, newRecipe), "Recipe1 should be edited");
    assertEquals("NewRecipe", newRecipe.getName(), "Recipe1 should be replaced by newRecipe");
}
```

### Purpose

This test verifies that the `editRecipe` method successfully replaces a recipe at a valid index and returns the name of the original recipe.

### Execution Report: FAILED

```
org.opentest4j.AssertionFailedError: Recipe1 should be replaced by newRecipe ==>
Expected :NewRecipe
Actual   :
```

## Test `RecipeBookTest::testEditRecipeInvalidIndexNegative`

### Test ID: 93

### Method

```
@Test
public void testEditRecipeInvalidIndexNegative() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        recipeBook.editRecipe(-1, newRecipe);
    }, "Negative index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test checks that the `editRecipe` method throws an `ArrayIndexOutOfBoundsException` when a negative index is provided.

### Execution Report: PASSED

## Test `RecipeBookTest::testEditRecipeInvalidIndexOutOfBounds`

### Test ID: 94

### Method

```java
@Test
public void testEditRecipeInvalidIndexOutOfBounds() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        recipeBook.editRecipe(4, newRecipe);
    }, "Out of bounds index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test verifies that the `editRecipe` method throws an `ArrayIndexOutOfBoundsException` when an out-of-bounds index is provided.

### Execution Report: PASSED

## Test `RecipeBookTest::testEditRecipeDoesNotExist`

### Test ID: 95

### Method

```java
@Test
public void testEditRecipeDoesNotExist() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertNull(recipeBook.editRecipe(0, newRecipe), "No recipe at index 0, should return null");
}
```

### Purpose

This test ensures that the `editRecipe` method returns `null` when attempting to edit a recipe at an empty index.

### Execution Report: PASSED

## Test `CoffeeMakerTest::testDefaultValues`

### Test ID: 96

### Method

```java
@Test
public void testDefaultValues() {
    // RecipeBook
    assertNotNull(coffeeMaker.getRecipes());
    assertEquals(4, coffeeMaker.getRecipes().length);

    // Inventory
    String inventory = coffeeMaker.checkInventory();
    assertTrue(inventory.contains("Coffee: 15"));
    assertTrue(inventory.contains("Milk: 15"));
    assertTrue(inventory.contains("Chocolate: 15"));
```

```
    assertTrue(inventory.contains("Sugar: 15"));
}
```

### Purpose

This test verifies that a newly created `CoffeeMaker` object initializes its `RecipeBook` with 4 empty recipe slots and its `Inventory` with default values (15 units of coffee, milk, sugar, and chocolate).

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testAddRecipeSuccess`

### Test ID: **97**

### Method

```
@Test
public void testAddRecipeSuccess() {
    assertTrue(coffeeMaker.addRecipe(recipe1), "Recipe1 should be added");
}
```

### Purpose

This test ensures that the `addRecipe` method successfully adds a valid recipe to the `CoffeeMaker`.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testAddRecipeDuplicate`

### Test ID: **98**

### Method

```
@Test
public void testAddRecipeDuplicate() {
    coffeeMaker.addRecipe(recipe1);
    assertFalse(coffeeMaker.addRecipe(recipe1), "Recipe1 should not be added again");
}
```

### Purpose

This test checks that the `addRecipe` method returns `false` when attempting to add a duplicate recipe.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testDeleteRecipeSuccess`

### Test ID: **99**

### Method

```
@Test
public void testDeleteRecipeSuccess() {
    coffeeMaker.addRecipe(recipe1);
    assertEquals("Recipe1", coffeeMaker.deleteRecipe(0), "Recipe1 should be deleted");

    Recipe[] recipes = coffeeMaker.getRecipes();
```

```
    assertNull(recipes[0], "Recipe1 deleted, so index 0 should be null");
}
```

### Purpose

This test verifies that the `deleteRecipe` method successfully deletes a recipe at a valid index and returns the name of the deleted recipe.

### Execution Report: `FAILED`

```
org.opentest4j.AssertionFailedError: Recipe1 deleted, so index 0 should be null ==>
Expected :null
Actual   :
```

## Test `CoffeeMakerTest::testDeleteRecipeInvalidIndexNegative`

### Test ID: `100`

### Method

```
@Test
public void testDeleteRecipeInvalidIndexNegative() {
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        coffeeMaker.deleteRecipe(-1);
    }, "Negative index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test ensures that the `deleteRecipe` method throws an `ArrayIndexOutOfBoundsException` when a negative index is provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testDeleteRecipeInvalidIndexOutOfBounds`

### Test ID: `101`

### Method

```
@Test
public void testDeleteRecipeInvalidIndexOutOfBounds() {
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        coffeeMaker.deleteRecipe(4);
    }, "Out of bounds index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test checks that the `deleteRecipe` method throws an `ArrayIndexOutOfBoundsException` when an out-of-bounds index is provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testDeleteRecipeDoesNotExist`

### Test ID: `102`

### Method

```java
@Test
public void testDeleteRecipeDoesNotExist() {
    assertNull(coffeeMaker.deleteRecipe(0), "No recipe at index 0, so should return null");
}
```

### Purpose

This test verifies that the `deleteRecipe` method returns `null` when attempting to delete a recipe from an empty index.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testEditRecipeSuccess`

### Test ID: `103`

### Method

```java
@Test
public void testEditRecipeSuccess() {
    coffeeMaker.addRecipe(recipe1);
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertEquals("Recipe1", coffeeMaker.editRecipe(0, newRecipe), "Recipe1 should be edited");
    assertEquals("NewRecipe", newRecipe.getName(), "Recipe1 should be replaced by newRecipe");
}
```

### Purpose

This test ensures that the `editRecipe` method successfully replaces a recipe at a valid index and returns the name of the original recipe.

### Execution Report: `FAILED`

```
org.opentest4j.AssertionFailedError: Recipe1 should be replaced by newRecipe ==>
Expected :NewRecipe
Actual   :
```

## Test `CoffeeMakerTest::testEditRecipeInvalidIndexNegative`

### Test ID: `104`

### Method

```java
@Test
public void testEditRecipeInvalidIndexNegative() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        coffeeMaker.editRecipe(-1, newRecipe);
    }, "Negative index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test checks that the `editRecipe` method throws an `ArrayIndexOutOfBoundsException` when a negative index is provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testEditRecipeInvalidIndexOutOfBounds`

### Test ID: `105`

### Method

```
@Test
public void testEditRecipeInvalidIndexOutOfBounds() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
        coffeeMaker.editRecipe(4, newRecipe);
    }, "Out of bounds index should throw ArrayIndexOutOfBoundsException");
}
```

### Purpose

This test verifies that the `editRecipe` method throws an `ArrayIndexOutOfBoundsException` when an out-of-bounds index is provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testEditRecipeDoesNotExist`

### Test ID: `106`

### Method

```
@Test
public void testEditRecipeDoesNotExist() {
    Recipe newRecipe = new Recipe();
    newRecipe.setName("NewRecipe");
    assertNull(coffeeMaker.editRecipe(0, newRecipe), "No recipe at index 0, so should return null");
}
```

### Purpose

This test ensures that the `editRecipe` method returns `null` when attempting to edit a recipe at an empty index.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testAddInventorySuccess`

### Test ID: `107`

### Method

```
@Test
public void testAddInventorySuccess() {
    try {
        coffeeMaker.addInventory("10", "10", "10", "10");
        String inventory = coffeeMaker.checkInventory();
        assertTrue(inventory.contains("Coffee: 25"));
        assertTrue(inventory.contains("Milk: 25"));
        assertTrue(inventory.contains("Sugar: 25"));
        assertTrue(inventory.contains("Chocolate: 25"));
    } catch (InventoryException e) {
        fail("Adding ingredient should not throw an exception");
    }
}
```

### Purpose

This test verifies that the `addInventory` method correctly increases the inventory amounts when valid inputs are provided.

### Execution Report: `FAILED`

```
coffee.exceptions.InventoryException: Units of sugar must be a positive integer
```

## Test `CoffeeMakerTest::testAddInventoryInvalidNonNumeric`

### Test ID: `108`

### Method

```
@Test
public void testAddInventoryInvalidNonNumeric() {
    assertThrows(InventoryException.class, () -> {
        coffeeMaker.addInventory("abc", "10", "10", "10");
    }, "Non-numeric input should throw InventoryException");
}
```

### Purpose

This test checks that the `addInventory` method throws an `InventoryException` when non-numeric inputs are provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testAddInventoryInvalidNegative`

### Test ID: `109`

### Method

```
@Test
public void testAddInventoryInvalidNegative() {
    assertThrows(InventoryException.class, () -> {
        coffeeMaker.addInventory("-5", "10", "10", "10");
    }, "Negative input should throw InventoryException");
}
```

### Purpose

This test ensures that the `addInventory` method throws an `InventoryException` when negative inputs are provided.

### Execution Report: `PASSED`

## Test `CoffeeMakerTest::testAddInventoryInvalidNull`

### Test ID: `110`

### Method

```
@Test
public void testAddInventoryInvalidNull() {
    assertThrows(InventoryException.class, () -> {
        coffeeMaker.addInventory(null, "10", "10", "10");
```

```
    }, "NULL input should throw InventoryException");
}
```

### Purpose

This test verifies that the `addInventory` method throws an `InventoryException` when `null` inputs are provided.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testMakeCoffeeSuccess`

### Test ID: **111**

### Method

```
@Test
public void testMakeCoffeeSuccess() {
    coffeeMaker.addRecipe(recipe1);
    int change = coffeeMaker.makeCoffee(0, 100);
    assertEquals(50, change, "Price is 50, paid 100. Change should be 50");
}
```

### Purpose

This test ensures that the `makeCoffee` method correctly calculates and returns the change when a valid recipe is selected and sufficient payment is provided.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testMakeCoffeeRecipeDoesNotExist`

### Test ID: **112**

### Method

```
@Test
public void testMakeCoffeeRecipeDoesNotExist() {
    int change = coffeeMaker.makeCoffee(0, 100);
    assertEquals(100, change, "Recipe does not exist, return full payment");
}
```

### Purpose

This test checks that the `makeCoffee` method returns the full payment when the selected recipe does not exist.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testMakeCoffeeInsufficientPayment`

### Test ID: **113**

### Method

```
@Test
public void testMakeCoffeeInsufficientPayment() {
    coffeeMaker.addRecipe(recipe1);
    int change = coffeeMaker.makeCoffee(0, 40);
```

```
    assertEquals(40, change, "Price is 50, paid 40 (insufficient), return 40");
}
```

### Purpose

This test verifies that the `makeCoffee` method returns the full payment when the payment is insufficient for the selected recipe.

### Execution Report: **PASSED**

## Test `CoffeeMakerTest::testMakeCoffeeInsufficientIngredients`

### Test ID: **114**

### Method

```
@Test
public void testMakeCoffeeInsufficientIngredients() {
    coffeeMaker.addRecipe(recipe2); // Requires 100 units of each ingredient
    int change = coffeeMaker.makeCoffee(0, 100);
    assertEquals(100, change, "Insufficient ingredients, return full payment");
}
```

### Purpose

This test ensures that the `makeCoffee` method returns the full payment when there are insufficient ingredients to make the selected recipe.

### Execution Report: **PASSED**

# Bug Report

## 1. Incorrect Inventory Initialization

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 1 | N/A | `Inventory::Inventory` | 18 |

**Description:** The `Inventory` constructor initializes the inventory with 15 units of each ingredient, but the fields are declared as `static`. This means that the inventory is shared across all instances of the `Inventory` class, which is not the intended behavior.

**Impact:** If multiple instances of the `Inventory` class are created, they will all share the same inventory, leading to incorrect inventory management.

**Recommended Fix:** Remove the `static` modifier from the inventory fields to ensure that each instance of the `Inventory` class has its own inventory.

```
private int coffee;
private int milk;
private int sugar;
private int chocolate;
```

## 2. Incorrect Inventory Update in `useIngredients` Method

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 2 | 77 | `Inventory::useIngredients` | 220 |

**Description:** The `useIngredients` method in the `Inventory` class incorrectly updates the inventory when ingredients are used. Specifically, the method adds the amount of `coffee` instead of subtracting it, which leads to an incorrect inventory count.

**Impact:** This bug will cause the inventory to increase when a recipe is made, which is incorrect. This will lead to inaccurate inventory tracking and potential issues when trying to make subsequent recipes.

**Recommended Fix:** Modify the `useIngredients` method to correctly subtract the ingredients used from the inventory.

```
public synchronized boolean useIngredients(Recipe r) {
    if (enoughIngredients(r)) {
        Inventory.coffee -= r.getAmtCoffee();  // Subtract coffee
        Inventory.milk -= r.getAmtMilk();
        Inventory.sugar -= r.getAmtSugar();
        Inventory.chocolate -= r.getAmtChocolate();
        return true;
    } else {
        return false;
    }
}
```

## 3. Incorrect Handling of Negative Values in `addSugar` Method

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 3 | 59, 61 | `Inventory::addSugar` | 182 |

**Description:** The `addSugar` method in the `Inventory` class incorrectly checks for negative values. The condition `if (amtSugar <= 0)` is used, which allows negative values to be added to the inventory and reject valid positive values. This can lead to incorrect inventory counts.

**Impact:** Negative values can be added to the sugar inventory, which can cause the inventory to become incorrect and potentially lead to issues when making recipes.

**Recommended Fix:** Modify the `addSugar` method to correctly check for positive values before adding them to the inventory.

```
public synchronized void addSugar(String sugar) throws InventoryException {
    int amtSugar = 0;
    try {
        amtSugar = Integer.parseInt(sugar);
    } catch (NumberFormatException e) {
        throw new InventoryException("Units of sugar must be a positive integer");
    }
    if (amtSugar >= 0) {
        Inventory.sugar += amtSugar;
    } else {
        throw new InventoryException("Units of sugar must be a positive integer");
    }
}
```

## 4. Incorrect Recipe Deletion Logic

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 4 | 88 | `RecipeBook::deleteRecipe` | 60 |

**Description:** The `deleteRecipe` method in the `RecipeBook` class incorrectly sets the recipe to a new empty `Recipe` object instead of setting it to `null`. This can lead to confusion and potential issues when checking for empty slots in the recipe array.

**Impact:** The recipe array will not be properly cleared, and the slot will still contain an empty `Recipe` object instead of being set to `null`. This can cause issues when adding new recipes or checking for available slots.

**Recommended Fix:** Modify the `deleteRecipe` method to set the recipe to `null` instead of creating a new `Recipe` object.

```
public synchronized String deleteRecipe(int recipeToDelete) {
    if (recipeArray[recipeToDelete] != null) {
        String recipeName = recipeArray[recipeToDelete].getName();
        recipeArray[recipeToDelete] = null; // Set to null instead of new Recipe()
        return recipeName;
    } else {
        return null;
    }
}
```

## 5. Incorrect Recipe Editing Logic

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 5 | 92, 103 | `RecipeBook::editRecipe` | 77 |

**Description:** The `editRecipe` method in the `RecipeBook` class incorrectly sets the name of the new recipe to an empty string before replacing the existing recipe. This can lead to the loss of the recipe name and potential issues when displaying or referencing the recipe.

**Impact:** The recipe name will be lost when editing a recipe, which can cause confusion and issues when trying to reference or display the recipe.

**Recommended Fix:** Modify the `editRecipe` method to preserve the name of the new recipe when replacing the existing recipe.

```
public synchronized String editRecipe(int recipeToEdit, Recipe newRecipe) {
    if (recipeArray[recipeToEdit] != null) {
        String recipeName = recipeArray[recipeToEdit].getName();
        recipeArray[recipeToEdit] = newRecipe; // Preserve the name of the new recipe
        return recipeName;
    } else {
        return null;
    }
}
```

## 6. Inflexible Recipe Array Size

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 6 | N/A | `RecipeBook::RecipeBook` | 13 |

**Description:** The `RecipeBook` constructor initializes the recipe array with a size of 4, but the `NUM_RECIPES` constant is set to 4. This means that the array size is hardcoded and not flexible.

**Impact:** If the number of recipes needs to be changed, the code must be modified, which is not ideal for maintainability.

**Recommended Fix:** Create an overloaded constructor that takes the number of recipes as a parameter to allow for a flexible array size.

```
public RecipeBook() {
    recipeArray = new Recipe[NUM_RECIPES];
}

public RecipeBook(int numRecipes) {
    recipeArray = new Recipe[numRecipes];
}
```

### 7. Inconsistent Synchronization in `CoffeeMaker` Class

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 7 | N/A | `CoffeeMaker::makeCoffee` | 30, 41, 52 |

**Description:** The `makeCoffee` method in the `CoffeeMaker` class is synchronized, but other methods that modify shared resources (e.g., `addRecipe`, `deleteRecipe`) are not. This can lead to race conditions in a multi-threaded environment.

**Impact:** Inconsistent synchronization can cause race conditions, leading to incorrect behavior when multiple threads access shared resources simultaneously.

**Recommended Fix:** Ensure that all methods that modify shared resources are properly synchronized.

```
public synchronized boolean addRecipe(Recipe r) {
    // Existing logic
}

public synchronized String deleteRecipe(int recipeToDelete) {
    // Existing logic
}

public synchronized String editRecipe(int recipeToEdit, Recipe r) {
    // Existing logic
}
```

### 8. Missing empty string Checks in `Recipe` Class

| Bug ID | TestCase ID | Location | Line |
|---|---|---|---|
| 8 | N/A | `Recipe::setName` | 127 |

**Description:** The `setName` method in the `Recipe` class does checks if it's null but does not check if the input `name` is empty string. This can lead to recipes with empty names, which may cause issues when displaying or referencing recipes.

**Impact:** Recipes with empty names can cause confusion and issues when trying to reference or display them.

**Recommended Fix:** Add empty checks in the `setName` method.

```
public void setName(String name) {
    if (name != null && !name.trim().isEmpty()) {
        this.name = name;
    } else {
        throw new IllegalArgumentException("Recipe name cannot be null or empty");
    }
}
```

# Test Coverage Report

This report summarizes the code coverage metrics for various classes in the project. The coverage percentages include metrics for **Class**, **Method**, **Branch**, and **Line** coverage. Below are the results:

## Coverage Overview

| Class | Class % | Method % | Branch % | Line % |
|---|---|---|---|---|
| **CoffeeMaker** | 100% (1/1) | 100% (8/8) | 100% (6/6) | 95.2% (20/21) |
| **Inventory** | 100% (1/1) | 100% (16/16) | 100% (26/26) | 100% (72/72) |
| **Recipe** | 100% (1/1) | 100% (16/16) | 80.8% (21/26) | 95.2% (60/63) |
| **RecipeBook** | 100% (1/1) | 100% (5/5) | 100% (16/16) | 100% (26/26) |

## Detailed Class Coverage Analysis

### Recipe

- **Class Coverage:** 100% (1/1)
- **Method Coverage:** 100% (16/16)
- **Branch Coverage:** 80.8% (21/26)
- **Line Coverage:** 95.2% (60/63)

The **Recipe** class has high coverage in terms of methods and class, but the branch coverage is slightly lower at 80.8%, indicating there might be some untested conditional branches. Line coverage is also very good at 95.2%.

#### Uncovered Branches

```
161         @Override
162         public int hashCode() {
163                 final int prime = 31;
164                 int result = 1;
165                 result = prime * result + ((name == null) ? 0 : name.hashCode());
166                 return result;
167         }
168 
169         @Override
170         public boolean equals(Object obj) {
171                 if (this == obj)
172                         return true;
173                 if (obj == null)
174                         return false;
175                 if (getClass() != obj.getClass())
176                         return false;
177                 final Recipe other = (Recipe) obj;
178                 if (name == null) {
179                         if (other.name != null)
180                                 return false;
181                 } else if (!name.equals(other.name))
182                         return false;
183                 return true;
184         }
185 
186 }
```

#### Improve Branch Coverage in `hashCode` Method

- Test the scenario where `name` is `null`.

#### Improve Branch Coverage in `equals` Method

- Test the scenario where `obj` is the instance of a different `class`.
- Test the scenario where `name` is `null` for `this` but not for `obj`.