UNIVERSITY OF GOTHENBURG

CHALMERS

Software quality and testing

Group 24
Yasamin Fazelidehkordi
Amin Mahmoudifard
Patrik Samcenko

Number of pages: 17
Date: 25-2-2023

Problem 1 - Unit Testing (60 Points)

Test Descriptions:
The following functionalities should be tested by the test suite:

- Coffee maker tests:
  - Add a recipe (testAddRecipe).
  - Delete a present recipe (testDeleteRecipe2).
  - Delete an absent recipe (testDeleteRecipe1).
  - Add to inventory (testAddInventory).
  - Make coffee with valid parameters (testMakeCoffee1).
  - Make coffee with insufficient funds (testMakeCoffee2).
  - Make coffee with insufficient ingredients (testMakeCoffee3).
  - Make coffee with a negative price (testNegativeArgumentsForMakeCoffee1).
  - Make a negative amount of coffee (testNegativeArgumentsForMakeCoffee2).
- Inventory tests:
  - Add a negative amount of chocolate (testAddChocolateException).
  - Add a negative amount of coffee (testAddCoffeeException).
  - Add a negative amount of milk (testAddMilkExcetpion).
  - Add a negative amount of sugar (testAddSugarException).
  - Add a recipe with not enough ingredients (testEnoughIngredients).
  - Use a recipe with no ingredients (testUseIngredients).
- Recipe book tests:
  - Add a recipe already in the book (testAddRecipe).
  - Edit an existing recipe (testEditRecipe1).
  - Edit a recipe in an empty recipe book (testEditRecipe2).
  - Delete a recipe (testDeleteRecipe2).
  - Delete a non-existent recipe (testDeleteRecipe1).
  - Get all recipes from the recipe book (testGetRecipes).
- Recipe tests:
  - Create a recipe that uses a negative amount of coffee (testSetAmtCoffeeExcetpion).
  - Create a recipe that uses a negative amount of milk (testSetAmtMilkException).
  - Create a recipe that uses a negative amount of chocolate (testSetAmtChocolateException).
  - Create a recipe with an invalid name (testInputNameValidity).
  - Add a recipe with a NULL name (testAssertNameNotNull).
  - Add a recipe with an invalid price (testInvalidPriceInput).
  - Add a recipe with a negative price (testSetPriceException).

Faults:

- The addRecipe method in CoffeeMaker class does not check if the recipe is null.
- addSugar method in in Inventory class does not make work properly because of this code: if(amtSugar >0)
- The makeCoffee function in the CoffeeMaker class does not check if invalid arguments(negative values) are passed.
- In the Recipe class the setName function does not check if the input name is valid.
- In the Recipe class the setName function does not check if the input name is null.
- The setPrice function in Recipe class does not check if invalid parameters are passed.
- In class coffeeMaker an empty recipe could be added to the list of recipes.
- When testing the addInventory function the amount of chocolate and sugar is not added correctly. The problem with sugar is as mentioned above because of the addSugar function in Inventory class in line 182 but we could not detect the problem with addChocolate function in Inventory class.

Corrections:

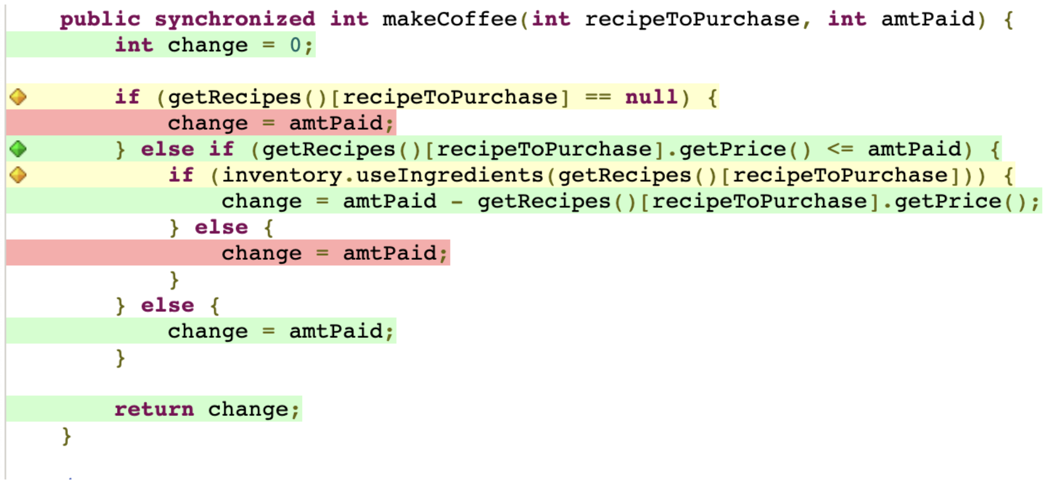
- Inventory class line 182 should be if (amtSugar <= 0)
- In the Recipe class add throw IllegalArgumentException to the setName function when the input name is not valid.
- In the Recipe class function add throw IllegalArgumentException to the setName function if the input price is not valid.
- In the Recipe class function, add “throw nullpointer exception” to the setName function for the null name inputs.
- In class coffeeMaker an empty recipe could be added to the list of recipes. It could be fixed by checking if the input recipe
- is empty or not in the addRecipe method in recipeBook class. We should add the following code:
```
if(r==null){
  System.out.println("The recipe is null and can not be added");
  added=false;
}
```

Problem 2 - Structural Coverage (40 Points)

Task1

CoffeeMaker->makeCoffee

Branch coverage before:

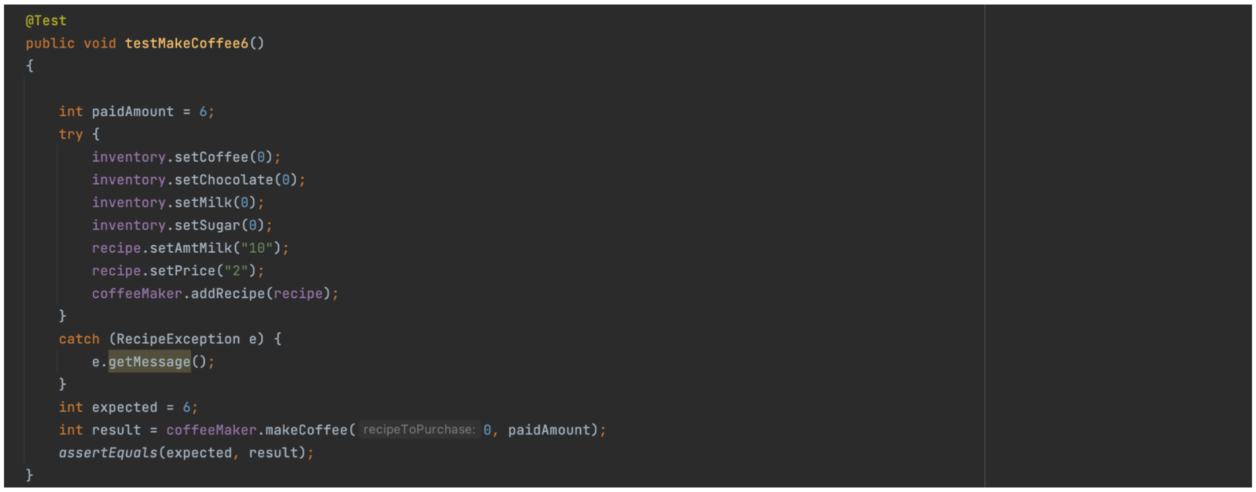


```
    public synchronized int makeCoffee(int recipeToPurchase, int amtPaid) {
        int change = 0;

        if (getRecipes()[recipeToPurchase] == null) {
            change = amtPaid;
        } else if (getRecipes()[recipeToPurchase].getPrice() <= amtPaid) {
            if (inventory.useIngredients(getRecipes()[recipeToPurchase])) {
                change = amtPaid - getRecipes()[recipeToPurchase].getPrice();
            } else {
                change = amtPaid;
            }
        } else {
            change = amtPaid;
        }

        return change;
    }
```

Added tests:

1)

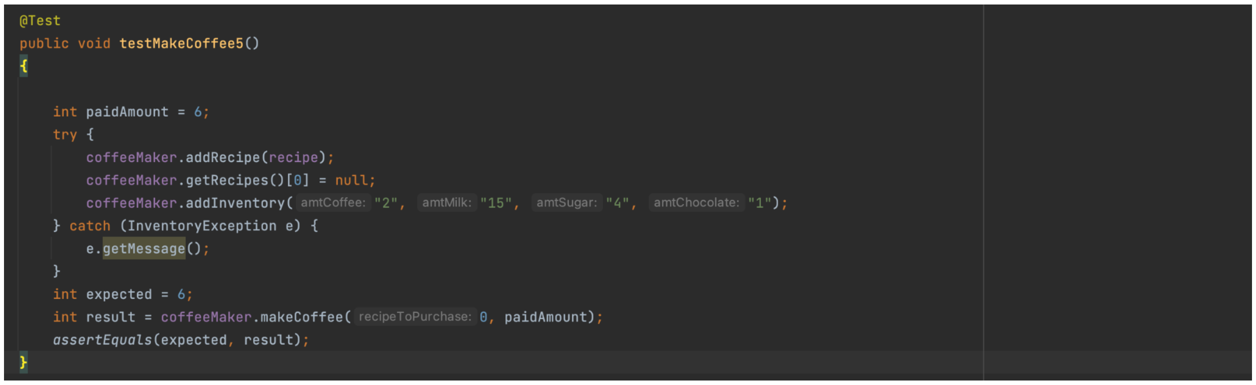
In this test what we're trying to cover in the code is the situation where our coffee machine doesn't have enough ingredients to be able to make the order. For that, we set all the ingredients in our machine to zero and set the required milk for our order to 10. Hence, the machine/code has to return the same amount that the user has entered back to us and we check that by assertEquals(expected, result).



```
@Test
public void testMakeCoffee6()
{

    int paidAmount = 6;
    try {
        inventory.setCoffee(0);
        inventory.setChocolate(0);
        inventory.setMilk(0);
        inventory.setSugar(0);
        recipe.setAmtMilk("10");
        recipe.setPrice("2");
        coffeeMaker.addRecipe(recipe);
    }
    catch (RecipeException e) {
        e.getMessage();
    }
    int expected = 6;
    int result = coffeeMaker.makeCoffee( recipeToPurchase: 0, paidAmount);
    assertEquals(expected, result);
}
```

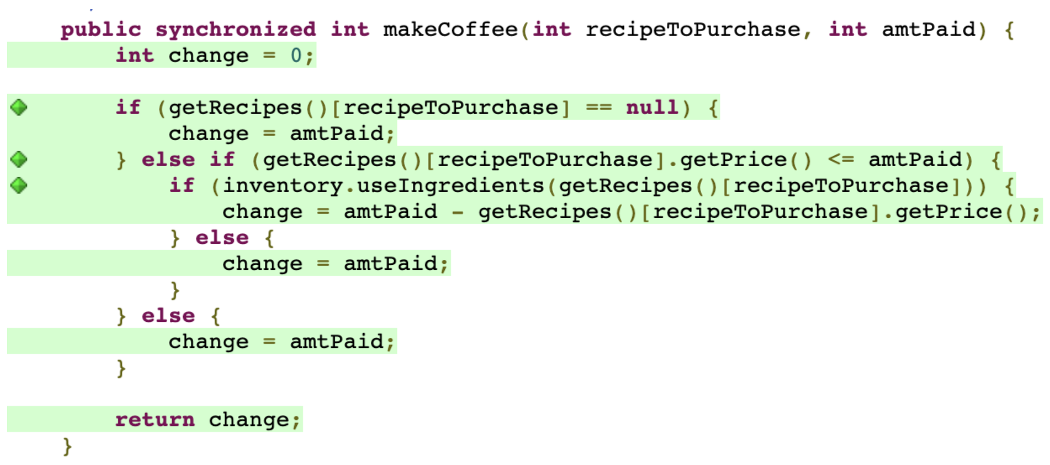
2)

In this test, we’re trying to cover/test the code in the case that the recipe being used is NULL. For that, we first add our recipe (in position 0) then make it NULL and ask for that recipe in position 0 which is NULL. When the recipe doesn’t exist, the machine/code has to return the same amount that the user has entered back to us and we check that by assertEquals(expected, result).



```java
@Test
public void testMakeCoffee5()
{

    int paidAmount = 6;
    try {
        coffeeMaker.addRecipe(recipe);
        coffeeMaker.getRecipes()[0] = null;
        coffeeMaker.addInventory( amtCoffee: "2",  amtMilk: "15",  amtSugar: "4",  amtChocolate: "1");
    } catch (InventoryException e) {
        e.getMessage();
    }
    int expected = 6;
    int result = coffeeMaker.makeCoffee( recipeToPurchase: 0, paidAmount);
    assertEquals(expected, result);
}
```

Branch coverage after:

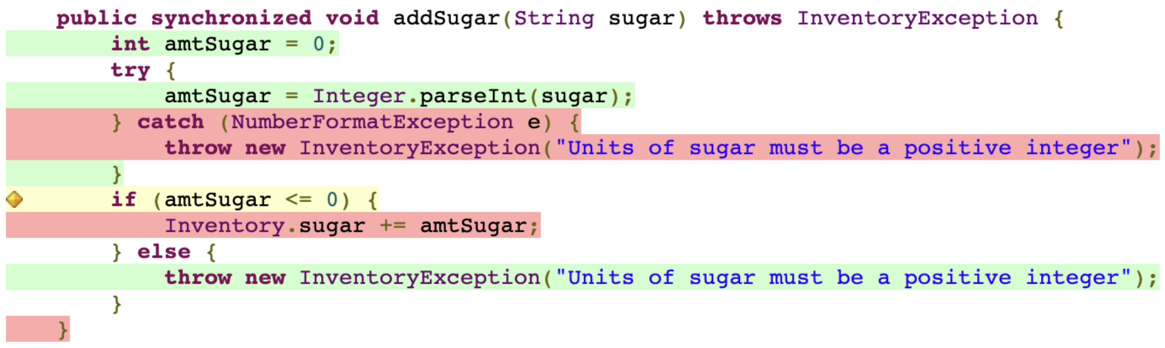


```java
public synchronized int makeCoffee(int recipeToPurchase, int amtPaid) {
    int change = 0;

    if (getRecipes()[recipeToPurchase] == null) {
        change = amtPaid;
    } else if (getRecipes()[recipeToPurchase].getPrice() <= amtPaid) {
        if (inventory.useIngredients(getRecipes()[recipeToPurchase])) {
            change = amtPaid - getRecipes()[recipeToPurchase].getPrice();
        } else {
            change = amtPaid;
        }
    } else {
        change = amtPaid;
    }

    return change;
}
```

As was noticeable in the picture of branch coverage before two cases of our if statements were not covered. Those are, when a recipe is NULL and when not enough ingredients exist in the machine to make an order. Those cases are now covered with the two additional tests (as shown in the picture above).

Inventory->addSugar

Branch coverage before:

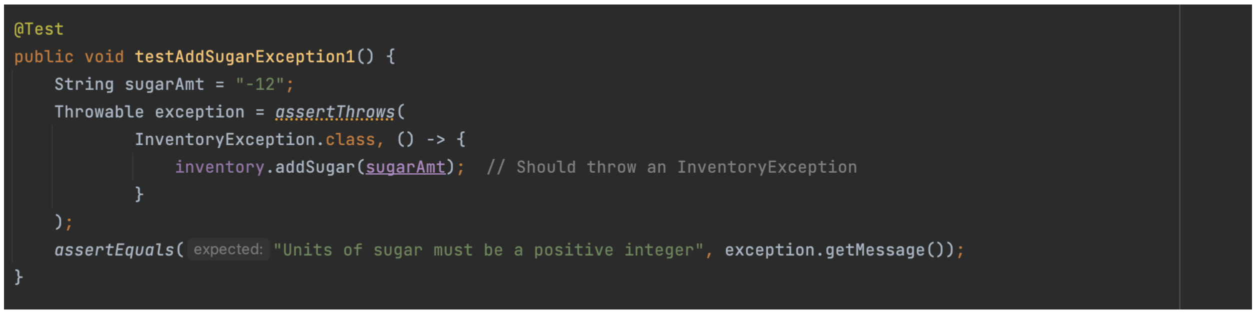


```java
    public synchronized void addSugar(String sugar) throws InventoryException {
        int amtSugar = 0;
        try {
            amtSugar = Integer.parseInt(sugar);
        } catch (NumberFormatException e) {
            throw new InventoryException("Units of sugar must be a positive integer");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

Added tests:

1)

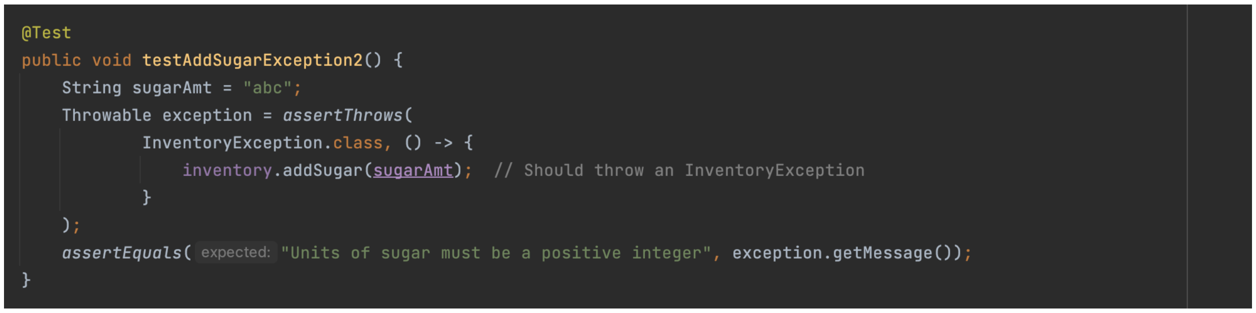
In the test below instead of concerting the string before passing it to the function, we pass it as it is so that we check that code either succeeds in converting or not. The code should return the error message but because of bugs in the code it doesn't and the test fails.



```java
@Test
public void testAddSugarException1() {
    String sugarAmt = "-12";
    Throwable exception = assertThrows(
            InventoryException.class, () -> {
                inventory.addSugar(sugarAmt);  // Should throw an InventoryException
            }
    );
    assertEquals( expected: "Units of sugar must be a positive integer", exception.getMessage());
}
```

2)

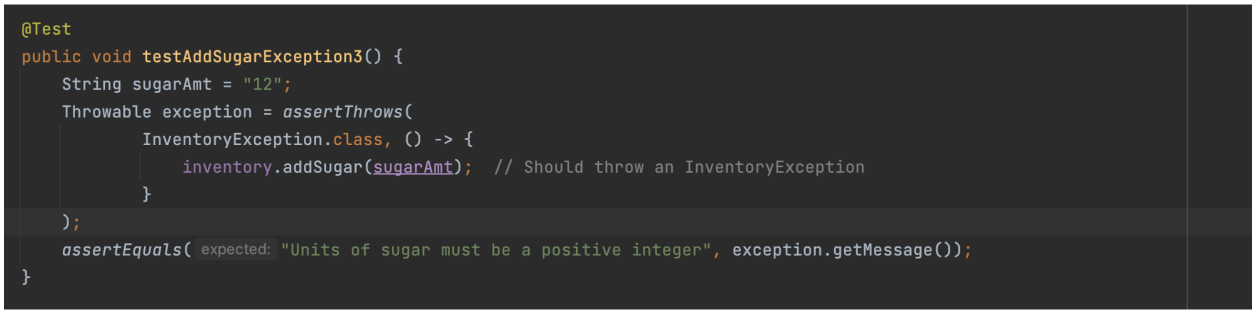
In the test below we consider the case where a string of characters (and not integers) is passed to the method since the method gets a string as a parameter. The code then crashes (throws an exception) when converting the string to an integer which makes us cover the catch block in the code.



```java
@Test
public void testAddSugarException2() {
    String sugarAmt = "abc";
    Throwable exception = assertThrows(
            InventoryException.class, () -> {
                inventory.addSugar(sugarAmt);  // Should throw an InventoryException
            }
    );
    assertEquals( expected: "Units of sugar must be a positive integer", exception.getMessage());
}
```

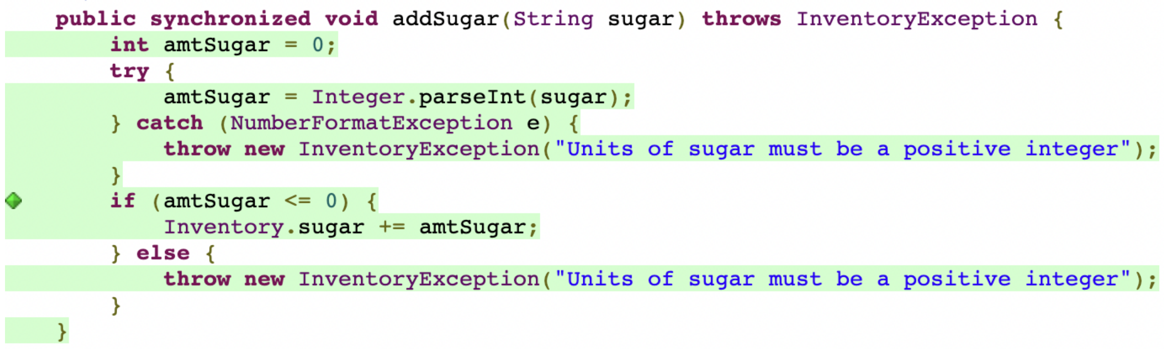
3)

In the previous tests, we have tried faulty cases/scenarios such as characters instead of integers and negative numbers. However because of the bug in the “if” statement, we have to test standard cases (positive integers) to achieve full branch coverage.



```java
@Test
public void testAddSugarException3() {
    String sugarAmt = "12";
    Throwable exception = assertThrows(
            InventoryException.class, () -> {
                inventory.addSugar(sugarAmt);  // Should throw an InventoryException
            }
    );
    assertEquals( expected: "Units of sugar must be a positive integer", exception.getMessage());
}
```

Branch coverage after:

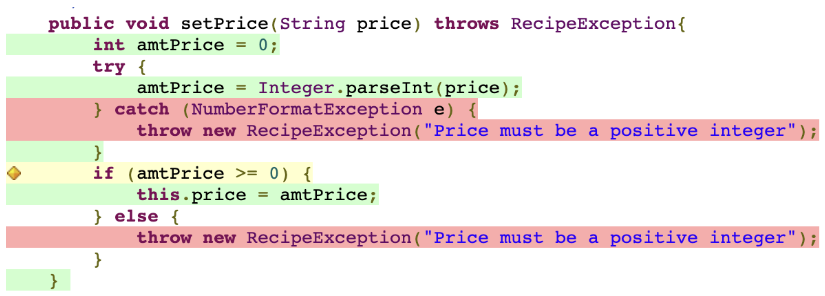


```java
    public synchronized void addSugar(String sugar) throws InventoryException {
        int amtSugar = 0;
        try {
            amtSugar = Integer.parseInt(sugar);
        } catch (NumberFormatException e) {
            throw new InventoryException("Units of sugar must be a positive integer");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

For this method, there were two cases (code parts) that were not covered. These were, the catch block and the actual “sugar adding” statement. By the tests above we tried to tiger the catch block by faulty/undesired strings and test the “sugar adding” function.

Recipe->setPrice

Branch coverage before:

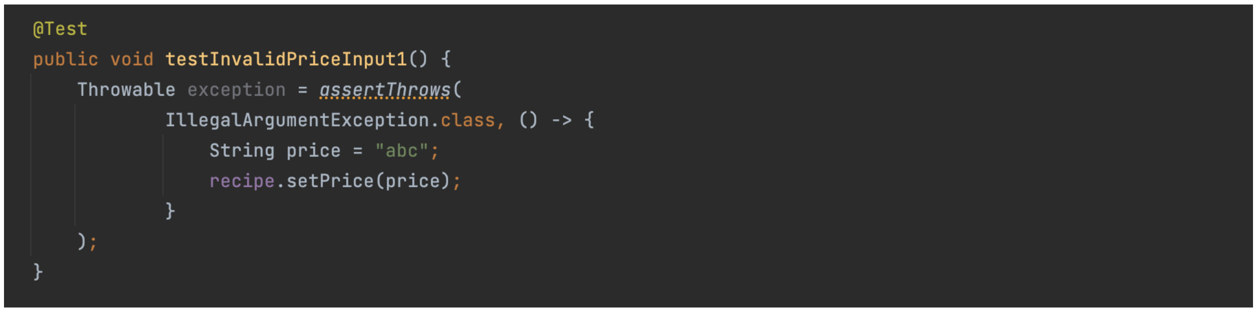


```java
    public void setPrice(String price) throws RecipeException{
        int amtPrice = 0;
        try {
            amtPrice = Integer.parseInt(price);
        } catch (NumberFormatException e) {
            throw new RecipeException("Price must be a positive integer");
        }
        if (amtPrice >= 0) {
            this.price = amtPrice;
        } else {
            throw new RecipeException("Price must be a positive integer");
        }
    }
```

Added tests:

1)

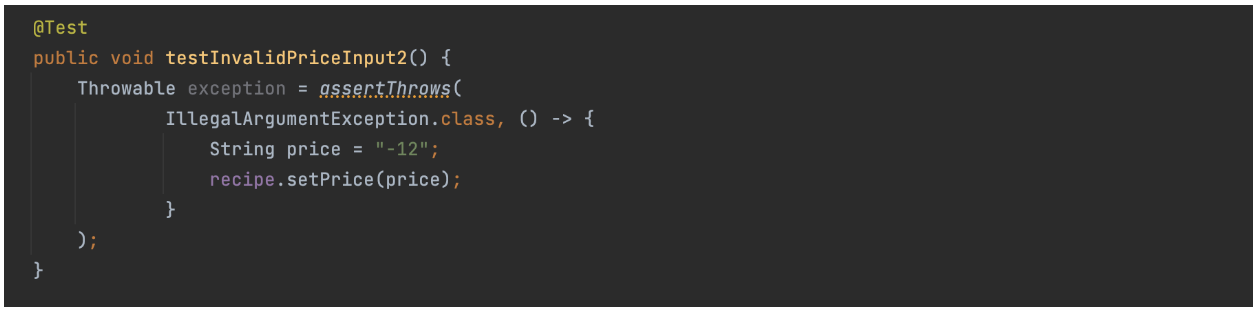
In this test we check the string converting functionality and try to cover its other branch being when it fails and throws an exception. For that, as a parameter, we pass “abc”. The catch block will be triggered and a RecipeException will be thrown to also cover that branch.



```java
@Test
public void testInvalidPriceInput1() {
    Throwable exception = assertThrows(
            IllegalArgumentException.class, () -> {
                String price = "abc";
                recipe.setPrice(price);
            }
    );
}
```

2)

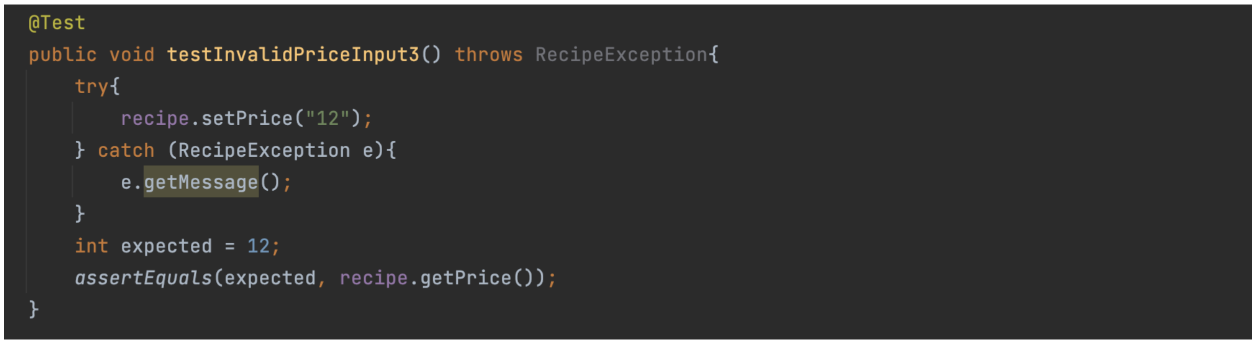
In this test we also cover the scenario where the passed price is negative. By doing so, the “else” statement will be triggered and a RecipeException will be thrown to also cover that branch.



```java
@Test
public void testInvalidPriceInput2() {
    Throwable exception = assertThrows(
            IllegalArgumentException.class, () -> {
                String price = "-12";
                recipe.setPrice(price);
            }
    );
}
```

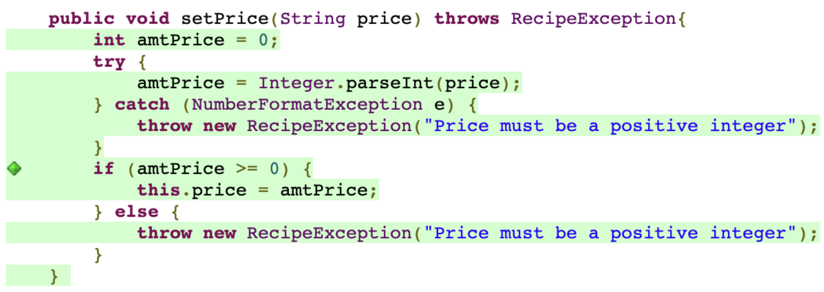
3)

In the test below we try to check the price declaration part of the code by trying to set the price and then comparing it with the expected number (which was 12).



```java
@Test
public void testInvalidPriceInput3() throws RecipeException{
    try{
        recipe.setPrice("12");
    } catch (RecipeException e){
        e.getMessage();
    }
    int expected = 12;
    assertEquals(expected, recipe.getPrice());
}
```

Branch coverage after:

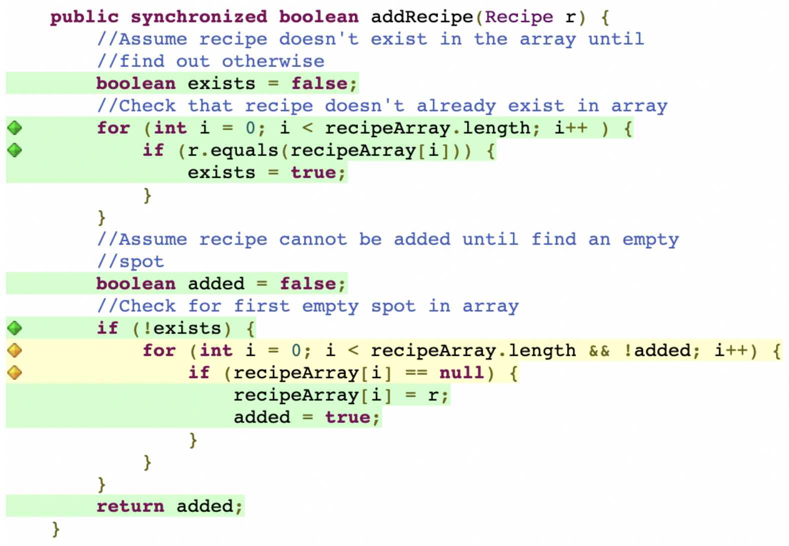


```java
    public void setPrice(String price) throws RecipeException{
        int amtPrice = 0;
        try {
            amtPrice = Integer.parseInt(price);
        } catch (NumberFormatException e) {
            throw new RecipeException("Price must be a positive integer");
        }
        if (amtPrice >= 0) {
            this.price = amtPrice;
        } else {
            throw new RecipeException("Price must be a positive integer");
        }
    }
```

This method was very much similar to the previous one and the same logic from the previous set of testings was applied to get full branch coverage.

RecipeBook->addRecipe

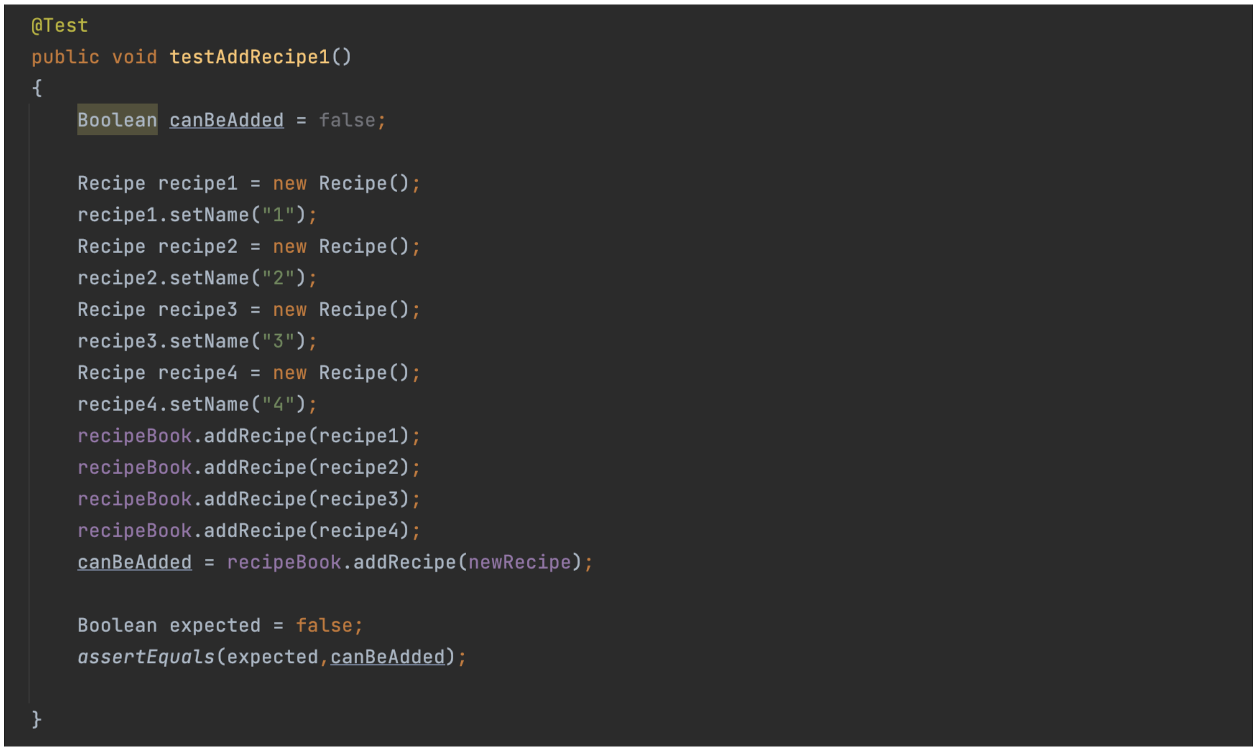
Branch coverage before:



```
    public synchronized boolean addRecipe(Recipe r) {
        //Assume recipe doesn't exist in the array until
        //find out otherwise
        boolean exists = false;
        //Check that recipe doesn't already exist in array
        for (int i = 0; i < recipeArray.length; i++ ) {
            if (r.equals(recipeArray[i])) {
                exists = true;
            }
        }
        //Assume recipe cannot be added until find an empty
        //spot
        boolean added = false;
        //Check for first empty spot in array
        if (!exists) {
            for (int i = 0; i < recipeArray.length && !added; i++) {
                if (recipeArray[i] == null) {
                    recipeArray[i] = r;
                    added = true;
                }
            }
        }
        return added;
    }
```

Added tests:

1)
In the test below we try to cover all the branches for the “for” loop by making the recipe book full so that in the code the boolean variable “added” never get true and we can go through the “for” loop for all the possible slots (1 - 4). We create four different recipes, add them and then add an additional one (which is basically what we test) and the result should be false since no extra slots are available in the recipe book, hence we cover all the branches for the “for” loop. Moreover, the case where `recipeArray[i] == null` was

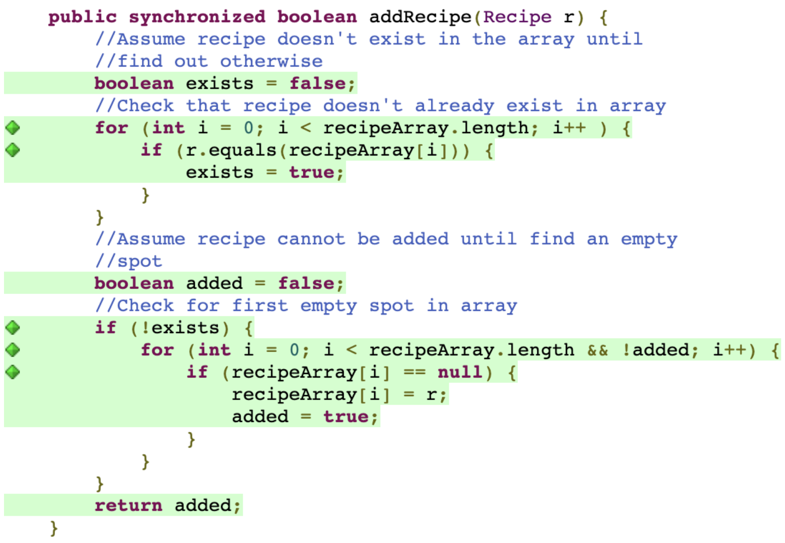


```
@Test
public void testAddRecipe1()
{
    Boolean canBeAdded = false;

    Recipe recipe1 = new Recipe();
    recipe1.setName("1");
    Recipe recipe2 = new Recipe();
    recipe2.setName("2");
    Recipe recipe3 = new Recipe();
    recipe3.setName("3");
    Recipe recipe4 = new Recipe();
    recipe4.setName("4");
    recipeBook.addRecipe(recipe1);
    recipeBook.addRecipe(recipe2);
    recipeBook.addRecipe(recipe3);
    recipeBook.addRecipe(recipe4);
    canBeAdded = recipeBook.addRecipe(newRecipe);

    Boolean expected = false;
    assertEquals(expected,canBeAdded);

}
```

not fully covered since it would always find a spot with a null value inside. Henceforth, this time round we also cover when it doesn't.

Branch coverage after:

For this method, only one more test was added to cover full branch coverage as it is explained above.



```
    public synchronized boolean addRecipe(Recipe r) {
        //Assume recipe doesn't exist in the array until
        //find out otherwise
        boolean exists = false;
        //Check that recipe doesn't already exist in array
        for (int i = 0; i < recipeArray.length; i++ ) {
            if (r.equals(recipeArray[i])) {
                exists = true;
            }
        }
        //Assume recipe cannot be added until find an empty
        //spot
        boolean added = false;
        //Check for first empty spot in array
        if (!exists) {
            for (int i = 0; i < recipeArray.length && !added; i++) {
                if (recipeArray[i] == null) {
                    recipeArray[i] = r;
                    added = true;
                }
            }
        }
        return added;
    }
```

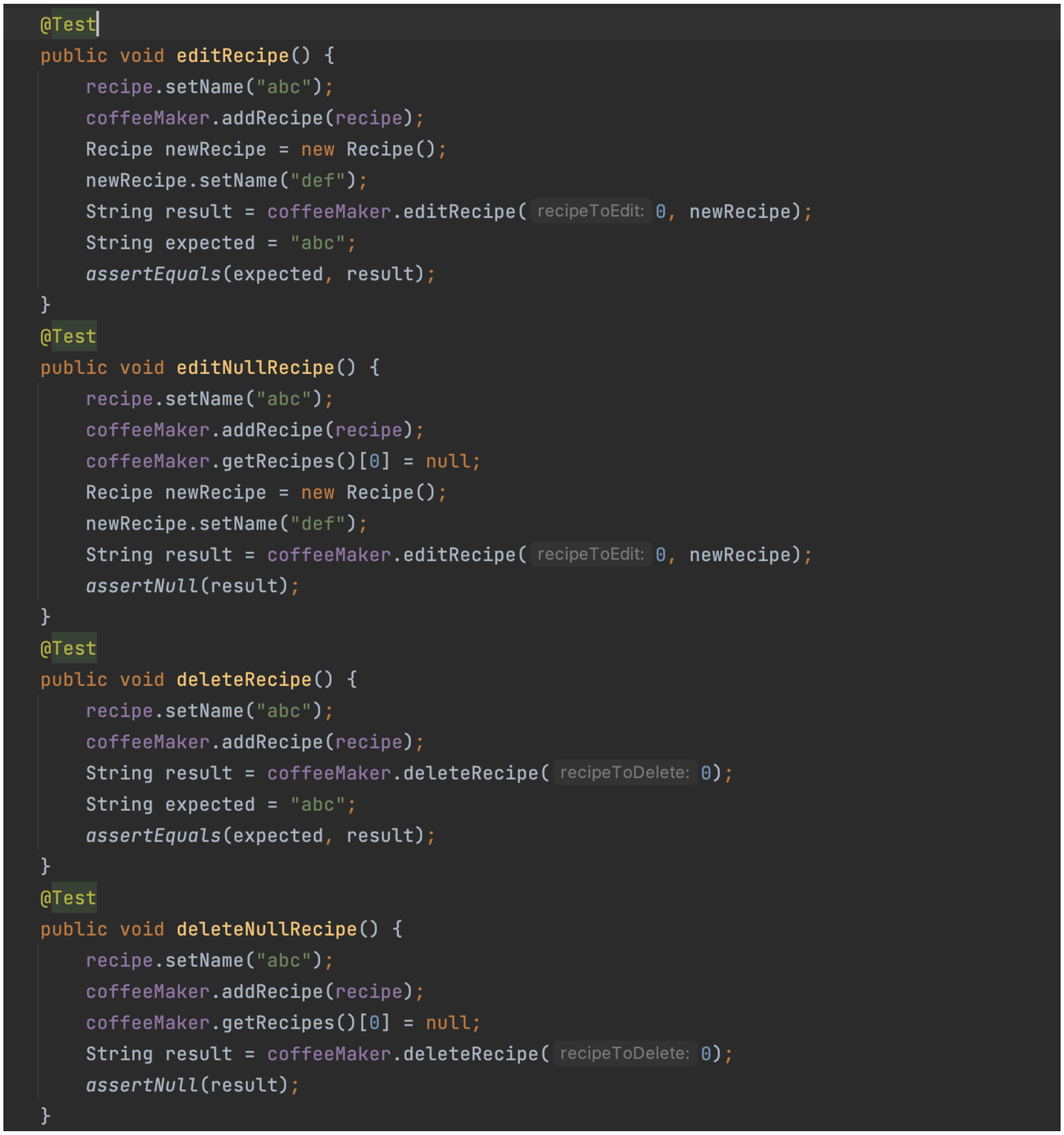
Task2

[Link to the line coverage report](s)

Task3

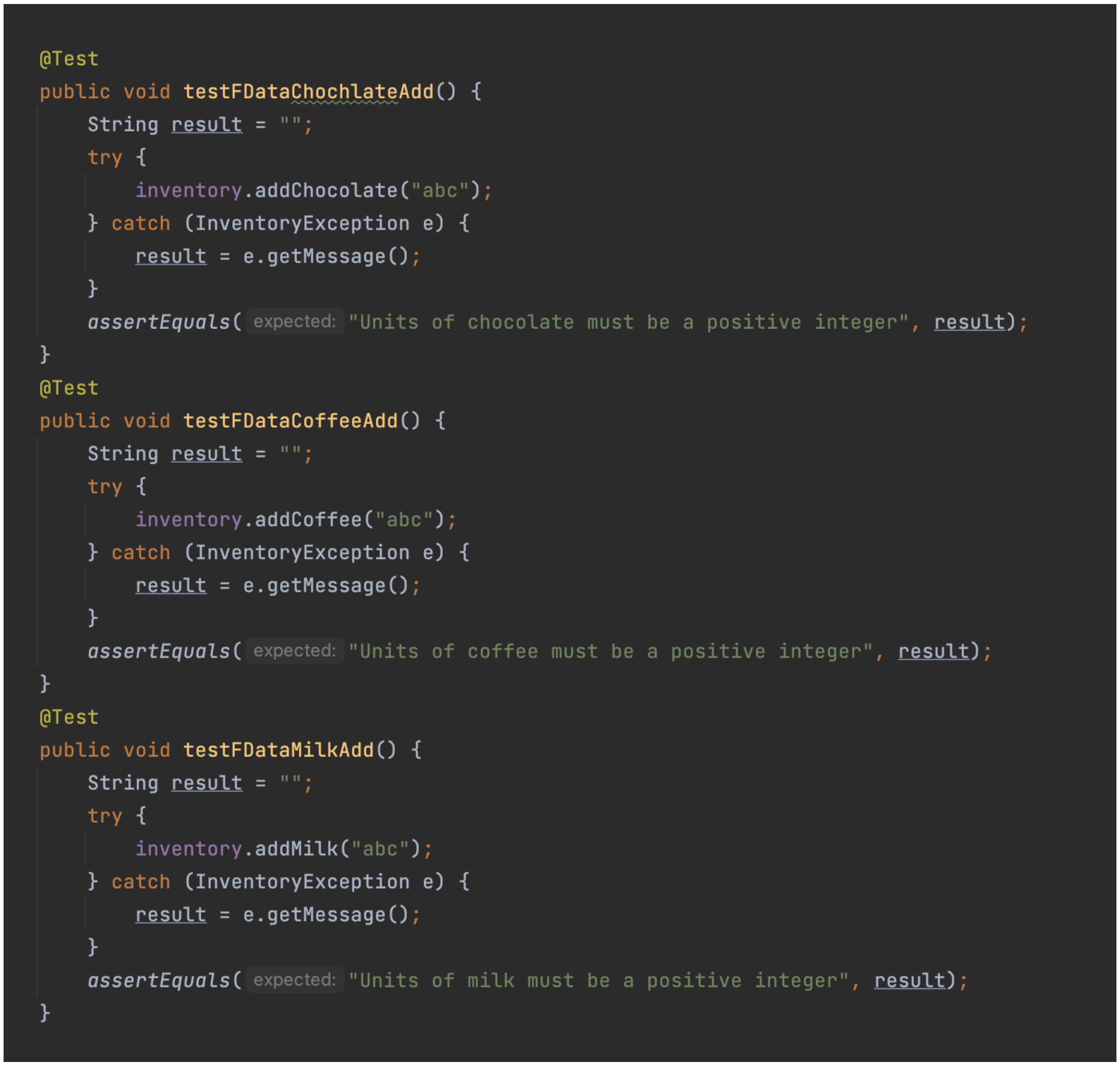
Added tests and why:

The additional tests for full line coverage for the coffee maker class are as follows. The edit and delete functions were fully ignored and we the tests below (editRecipe & deleteRecipe) we cover their functionalities which are basically editing the name of a recipe and deleting a recipe. Both of these methods if successful, return back the name of the recipe they are operated on. Moreover, the tests mentioned above do not cover the case when we try to operate on a NULL recipe. For that, we try to do the same pair of tests but this time the recipe that we're trying to work on doesn't exist. Both of these methods shall send a null value in this situation which our tests (editNullRecipe & deleteNullRecipe) check for them.



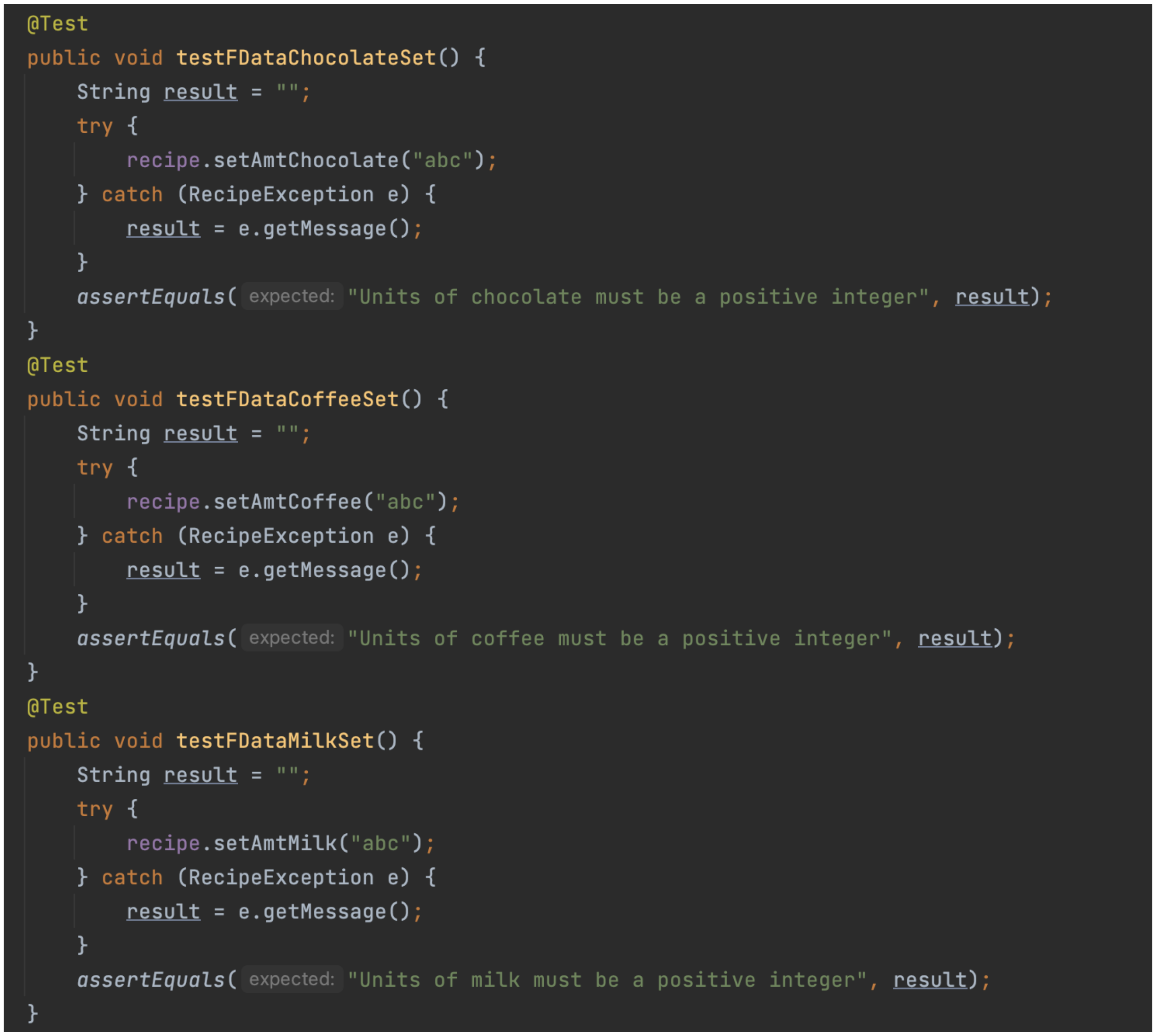
```java
@Test
public void editRecipe() {
    recipe.setName("abc");
    coffeeMaker.addRecipe(recipe);
    Recipe newRecipe = new Recipe();
    newRecipe.setName("def");
    String result = coffeeMaker.editRecipe( recipeToEdit: 0, newRecipe);
    String expected = "abc";
    assertEquals(expected, result);
}
@Test
public void editNullRecipe() {
    recipe.setName("abc");
    coffeeMaker.addRecipe(recipe);
    coffeeMaker.getRecipes()[0] = null;
    Recipe newRecipe = new Recipe();
    newRecipe.setName("def");
    String result = coffeeMaker.editRecipe( recipeToEdit: 0, newRecipe);
    assertNull(result);
}
@Test
public void deleteRecipe() {
    recipe.setName("abc");
    coffeeMaker.addRecipe(recipe);
    String result = coffeeMaker.deleteRecipe( recipeToDelete: 0);
    String expected = "abc";
    assertEquals(expected, result);
}
@Test
public void deleteNullRecipe() {
    recipe.setName("abc");
    coffeeMaker.addRecipe(recipe);
    coffeeMaker.getRecipes()[0] = null;
    String result = coffeeMaker.deleteRecipe( recipeToDelete: 0);
    assertNull(result);
}
```

Moving on to the Inventory class. The missing lines in this class were for addChocolate, addCoffee, and addMilk. The matter of if we pass in characters and integers was not covered in any of them. The tests below are basically the same test. They all try to pass in characters as the amount of ingredients we have to add to our inventory and they all throw InventoryException that we check for its accuracy.

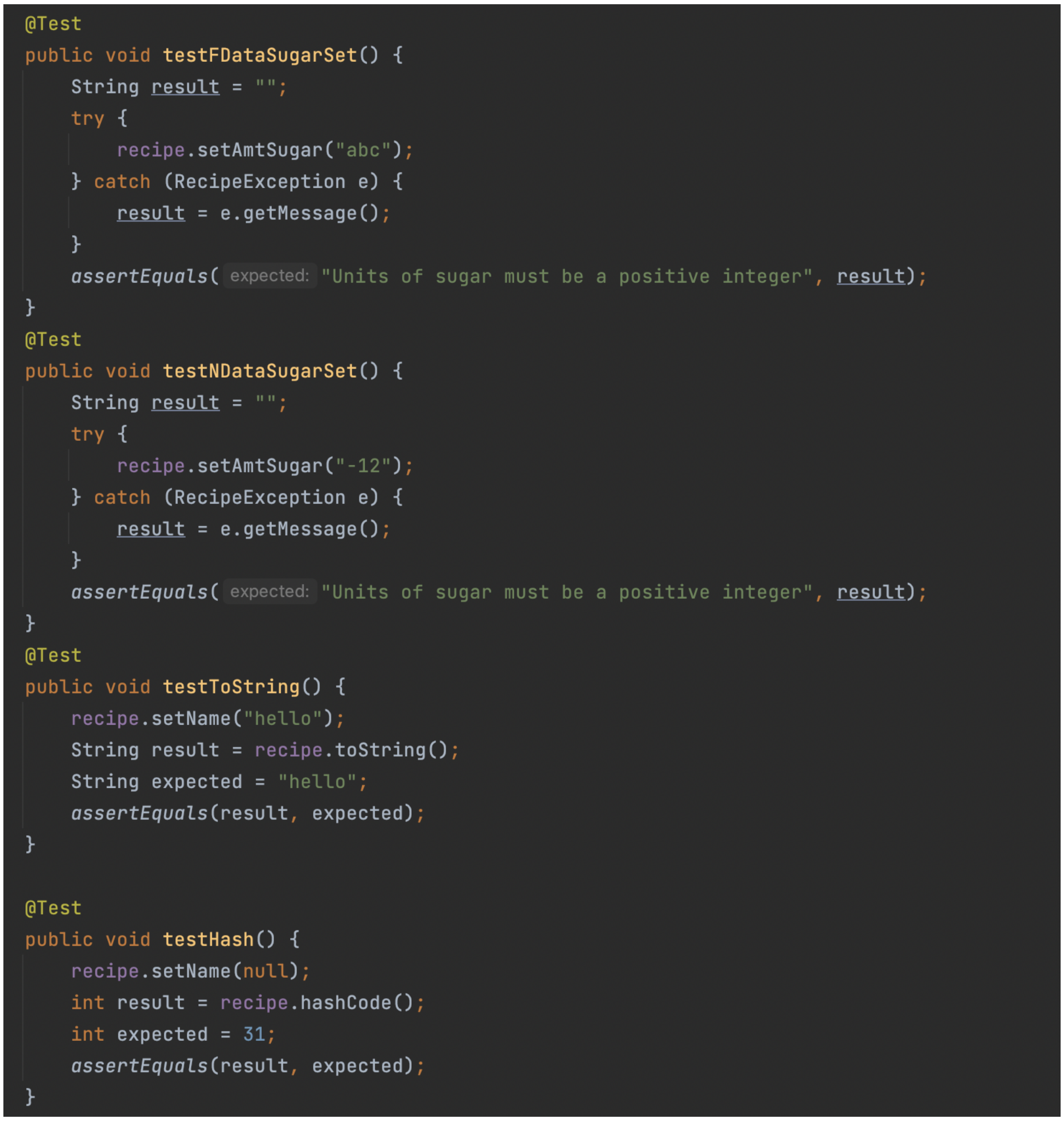


```java
@Test
public void testFDataChochlateAdd() {
    String result = "";
    try {
        inventory.addChocolate("abc");
    } catch (InventoryException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of chocolate must be a positive integer", result);
}
@Test
public void testFDataCoffeeAdd() {
    String result = "";
    try {
        inventory.addCoffee("abc");
    } catch (InventoryException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of coffee must be a positive integer", result);
}
@Test
public void testFDataMilkAdd() {
    String result = "";
    try {
        inventory.addMilk("abc");
    } catch (InventoryException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of milk must be a positive integer", result);
}
```

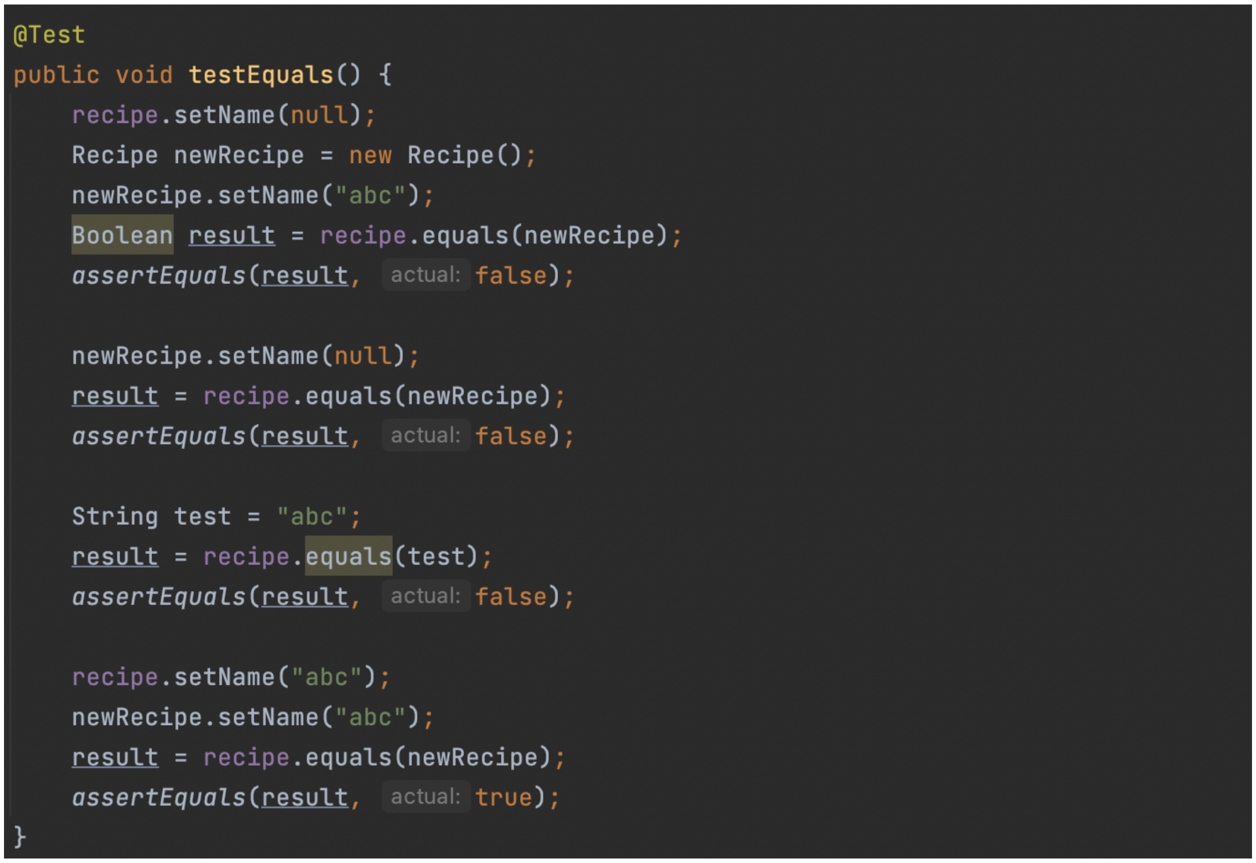
At last, we get to the Recipe class. The first four tests (testFdataChocolateSet, testFdataCoffeeSet, testFdataMilkSet, testFdataSugarSet) test the methods setAmtChocolate, setAmtCoffee, setAmtMilk, and setAmtSugar respectively. The only parts/lines that were missing in these methods were very similar to the ones in the previous class. As before the case where we pass in characters as integers was not covered by any of them. The first four tests below are basically the same test. They all try to pass in characters as the amount of ingredients we have to set and they all throw RecipeException that we check for its accuracy. Moreover, the testNdataSugarSet test is an additional one for the setAmtSugar method for the case we pass in negative integers as the amount of sugar we have to set. By doing so, the method should throw an exception which we check for its accuracy. Then, we test the toString function of the recipe class. For that, we create a recipe, set a name for it, and compare the name against the value the toString method returns for the recipe. Moreover, another test for the hashCode method is implemented to cover the accuracy of the calculations inside and at last a test for checking the equals method of the recipe class. For that, we test different combinations of cases, such as null-name for R1 - name for R2 / null-name for R1 - null-name for R2 / null-name for R1 - a string and lastly name for R1 - name for R2. For each of these, the expected results are false, false, false, and true respectively.



```
@Test
public void testFDataChocolateSet() {
    String result = "";
    try {
        recipe.setAmtChocolate("abc");
    } catch (RecipeException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of chocolate must be a positive integer", result);
}
@Test
public void testFDataCoffeeSet() {
    String result = "";
    try {
        recipe.setAmtCoffee("abc");
    } catch (RecipeException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of coffee must be a positive integer", result);
}
@Test
public void testFDataMilkSet() {
    String result = "";
    try {
        recipe.setAmtMilk("abc");
    } catch (RecipeException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of milk must be a positive integer", result);
}
```



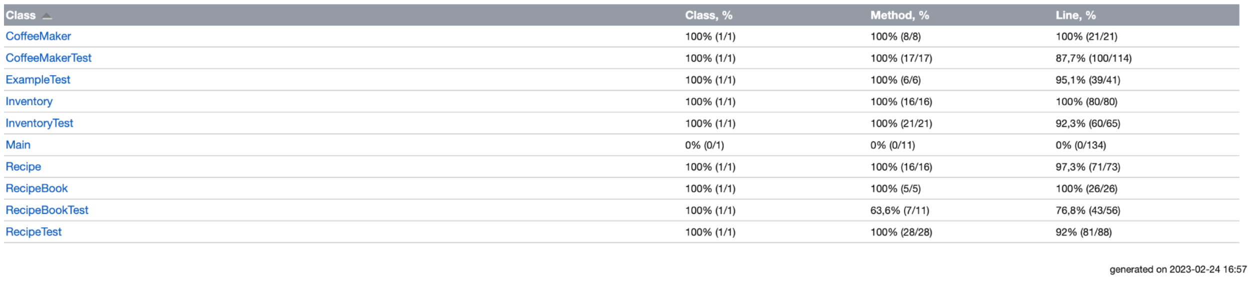
```
@Test
public void testFDataSugarSet() {
    String result = "";
    try {
        recipe.setAmtSugar("abc");
    } catch (RecipeException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of sugar must be a positive integer", result);
}
@Test
public void testNDataSugarSet() {
    String result = "";
    try {
        recipe.setAmtSugar("-12");
    } catch (RecipeException e) {
        result = e.getMessage();
    }
    assertEquals( expected: "Units of sugar must be a positive integer", result);
}
@Test
public void testToString() {
    recipe.setName("hello");
    String result = recipe.toString();
    String expected = "hello";
    assertEquals(result, expected);
}

@Test
public void testHash() {
    recipe.setName(null);
    int result = recipe.hashCode();
    int expected = 31;
    assertEquals(result, expected);
}
```



```java
@Test
public void testEquals() {
    recipe.setName(null);
    Recipe newRecipe = new Recipe();
    newRecipe.setName("abc");
    Boolean result = recipe.equals(newRecipe);
    assertEquals(result,  actual: false);

    newRecipe.setName(null);
    result = recipe.equals(newRecipe);
    assertEquals(result,  actual: false);

    String test = "abc";
    result = recipe.equals(test);
    assertEquals(result,  actual: false);

    recipe.setName("abc");
    newRecipe.setName("abc");
    result = recipe.equals(newRecipe);
    assertEquals(result,  actual: true);
}
```

Why not 100%?

As shown in the report below (the full report can also be accessed through the link provided in task2), all the classes (those that you've asked for) have 100% line coverage except the Recipe class which has 97.3% line coverage and the reason behind it is that in the equals method in the recipe class, there's a line that is not covered or better to say couldn't be covered. That specific line can only be reached when the recipe that you're comparing against another one has a NULL name. Nonetheless, that couldn't be set in the test since the name attribute of the recipe class is private and can't be set directly so you would have to use the setName method which only would allow setting names when that name you're trying to set is not NULL. To summarize, in order to reach that missed line, we had to set the name as NULL which wasn't possible.



Class	Class, %	Method, %	Line, %
CoffeeMaker	100% (1/1)	100% (8/8)	100% (21/21)
CoffeeMakerTest	100% (1/1)	100% (17/17)	87,7% (100/114)
ExampleTest	100% (1/1)	100% (6/6)	95,1% (39/41)
Inventory	100% (1/1)	100% (16/16)	100% (80/80)
InventoryTest	100% (1/1)	100% (21/21)	92,3% (60/65)
Main	0% (0/1)	0% (0/11)	0% (0/134)
Recipe	100% (1/1)	100% (16/16)	97,3% (71/73)
RecipeBook	100% (1/1)	100% (5/5)	100% (26/26)
RecipeBookTest	100% (1/1)	63,6% (7/11)	76,8% (43/56)
RecipeTest	100% (1/1)	100% (28/28)	92% (81/88)

generated on 2023-02-24 16:57