DIT635 Software Quality and Testing

Assignment 2: Unit and Structural Testing

Group 20:
Erdem Halil
George-Vlad Liteanu
Taofik Arnouk

27/02/2022

13 Pages

Problem 1 - Unit Testing (60 Points)

1. Test Descriptions.

Test description:
We describe the code in the same order as it appears below.

Unit test 1: Firstly we set up the file and add some recipes to the system.
This test tests if the “checkInventory” function functions correctly and if we receive the correct response by using “assertEquals(expected, inventory)” to check if the expected response is the same as the actual response received.

Unit test 2: This test tests if the most important function of the system is working correctly, that is the “makeCoffee” function. We call this function for the second recipe and check if the amount of change received is correct. We then also check if the number of items left in the inventory is correct.

Unit test 3: In this test, we create a correct, new recipe and try to add it to the system. We then check if the length of the array has increased and then we check if the recipes are the same.

Unit test 4: In this test, we try to add an invalid recipe to the system. As a response, we expect an error.

Unit test 5: In this test, we try to edit one of the recipes. We first create a new recipe and then we call the “editRecipe” method to replace an existing recipe. We then check if the added recipe has the correct values.

Unit test 6: In this test, we will try to edit a recipe and add invalid data to the system. We try to replace an existing recipe and expect an error.

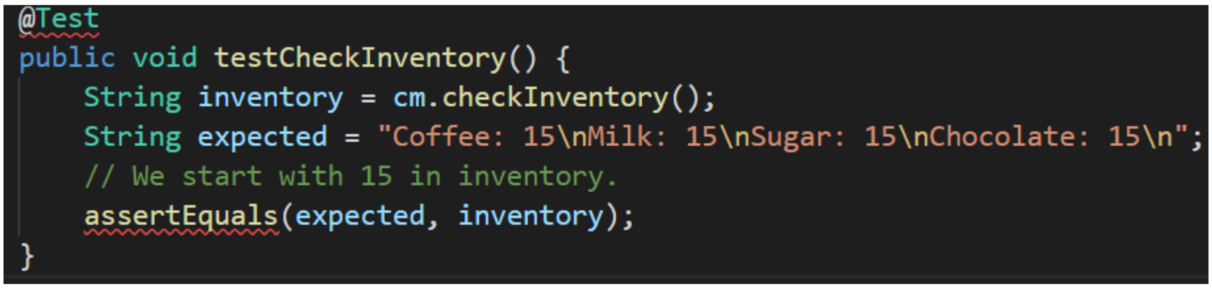
Unit test 7: This test will loop through all the recipes and try to delete all of them. It will then check that they are empty.

Unit test 8: This test will try to add items to the inventory and then it will check the number of items after the addition.

Unit test 9: This test will try to add invalid values of items to the inventory. We expect an error.

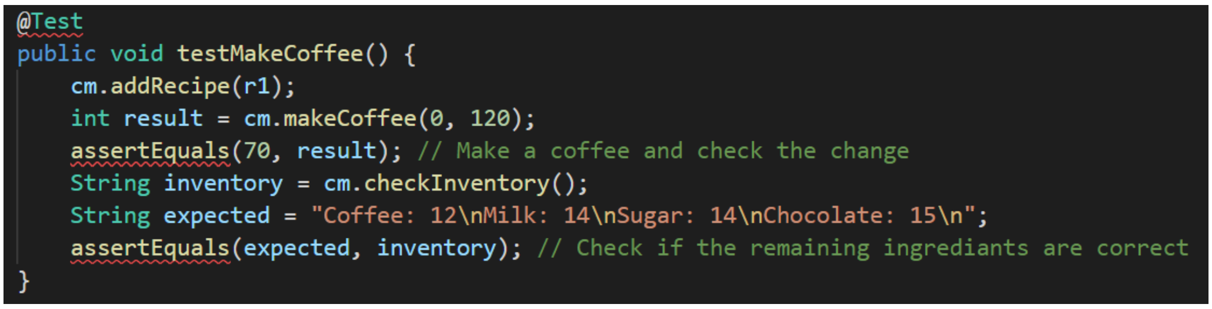
2. Unit tests implemented in the JUnit framework.

1



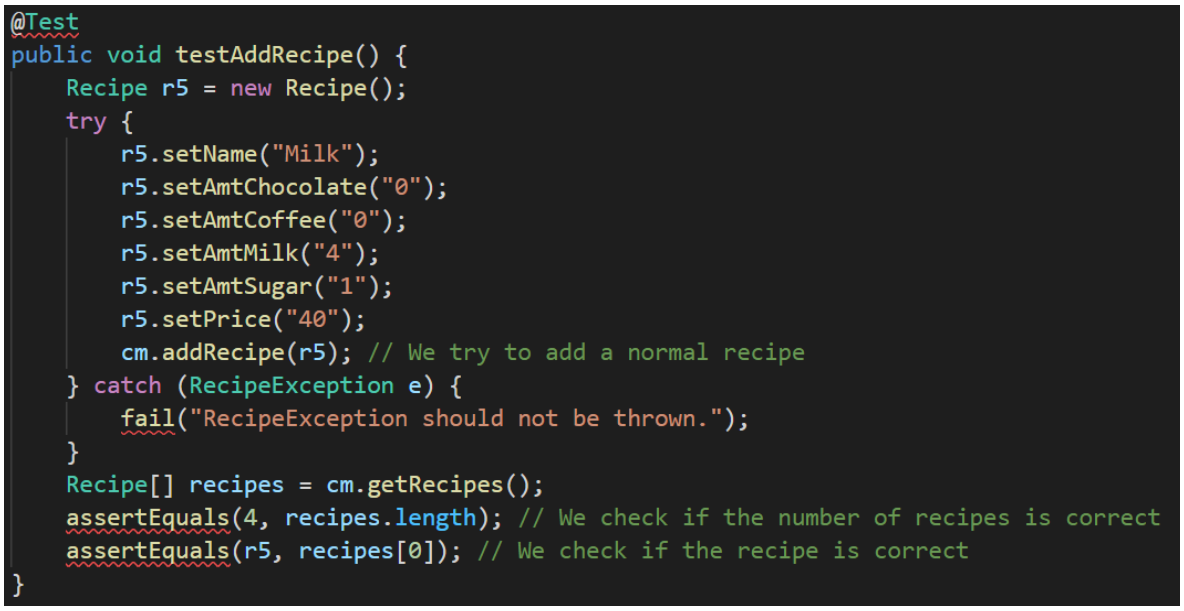
```java
@Test
public void testCheckInventory() {
    String inventory = cm.checkInventory();
    String expected = "Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n";
    // We start with 15 in inventory.
    assertEquals(expected, inventory);
}
```

2



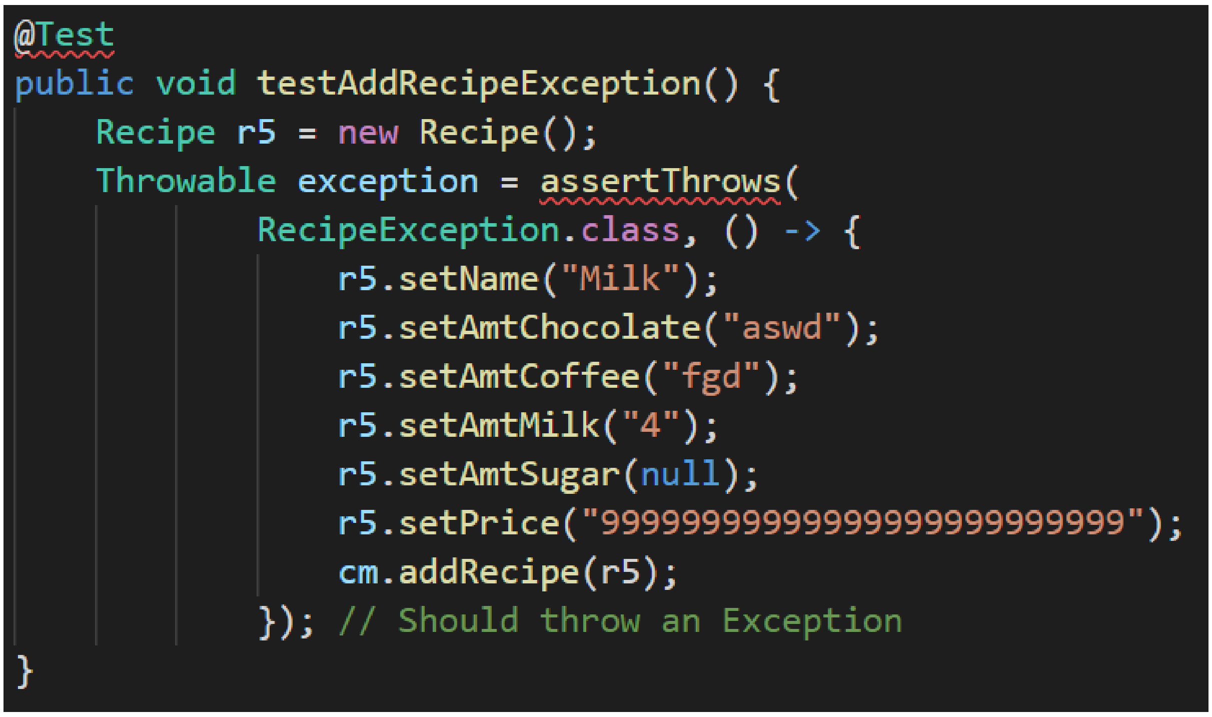
```java
@Test
public void testMakeCoffee() {
    cm.addRecipe(r1);
    int result = cm.makeCoffee(0, 120);
    assertEquals(70, result); // Make a coffee and check the change
    String inventory = cm.checkInventory();
    String expected = "Coffee: 12\nMilk: 14\nSugar: 14\nChocolate: 15\n";
    assertEquals(expected, inventory); // Check if the remaining ingrediants are correct
}
```

3



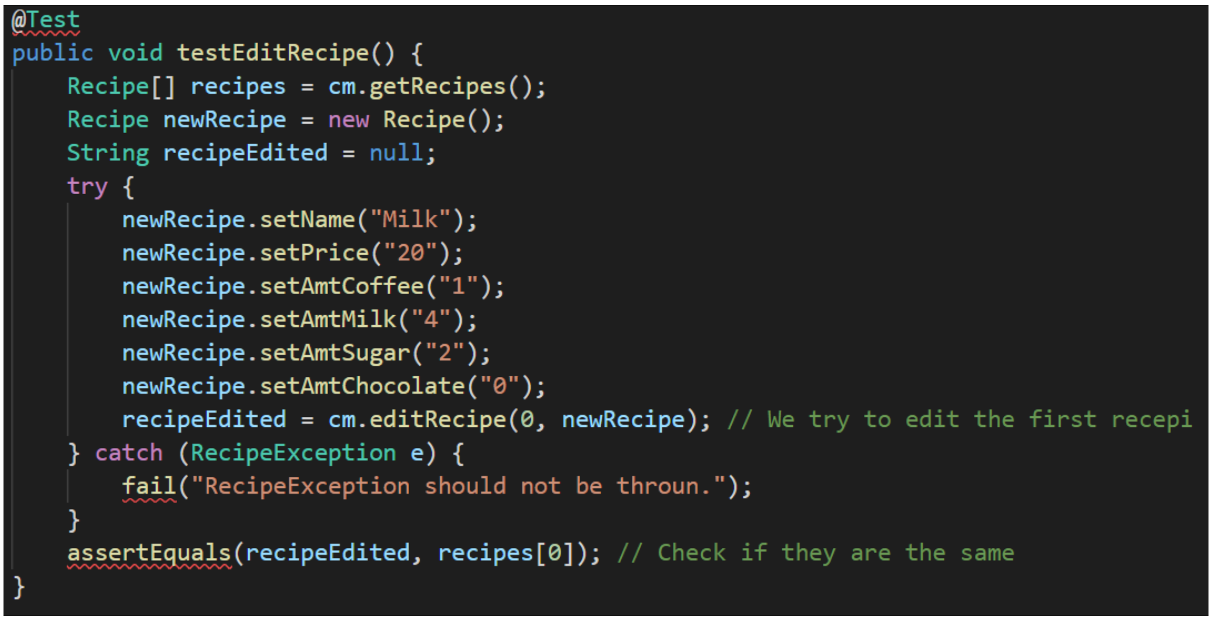
```java
@Test
public void testAddRecipe() {
    Recipe r5 = new Recipe();
    try {
        r5.setName("Milk");
        r5.setAmtChocolate("0");
        r5.setAmtCoffee("0");
        r5.setAmtMilk("4");
        r5.setAmtSugar("1");
        r5.setPrice("40");
        cm.addRecipe(r5); // We try to add a normal recipe
    } catch (RecipeException e) {
        fail("RecipeException should not be thrown.");
    }
    Recipe[] recipes = cm.getRecipes();
    assertEquals(4, recipes.length); // We check if the number of recipes is correct
    assertEquals(r5, recipes[0]); // We check if the recipe is correct
}
```

4



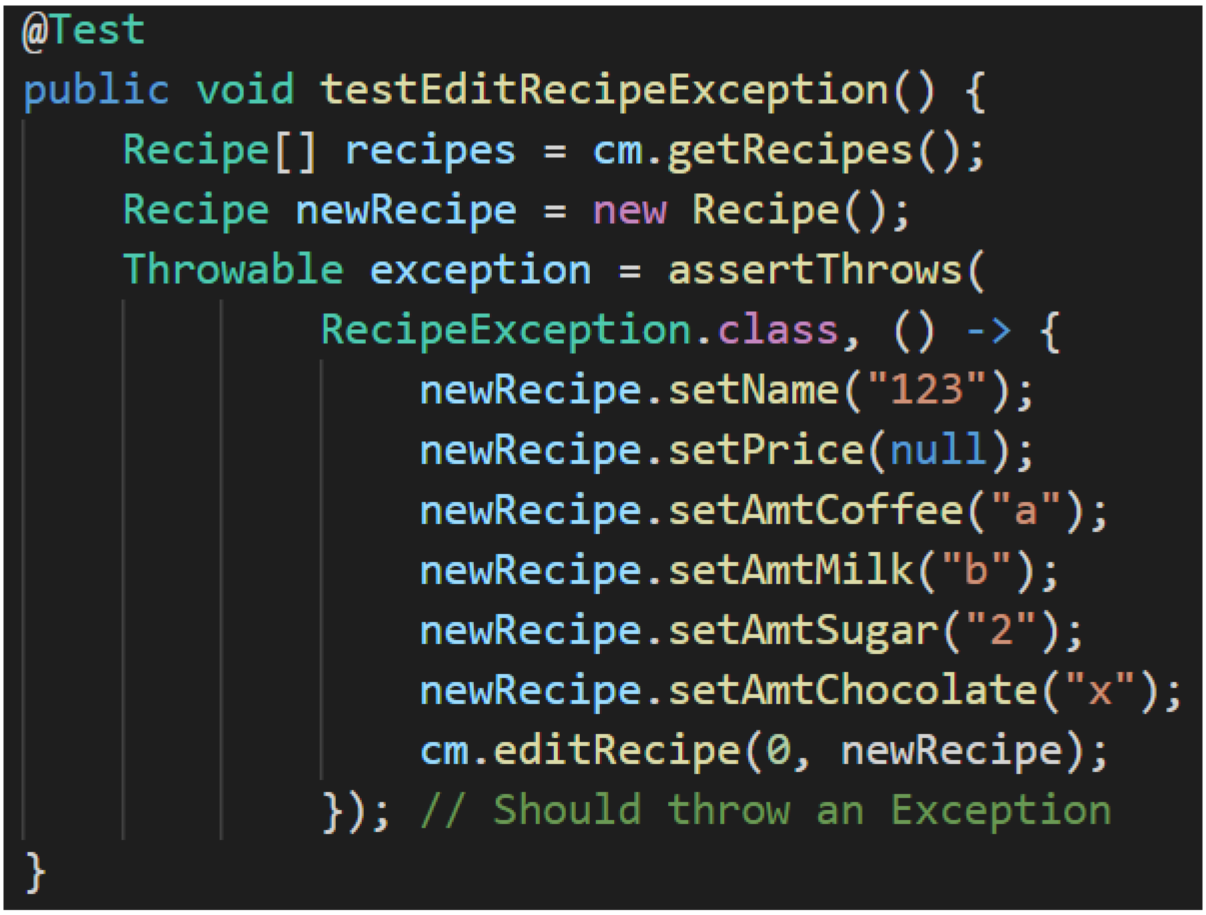
```
@Test
public void testAddRecipeException() {
    Recipe r5 = new Recipe();
    Throwable exception = assertThrows(
            RecipeException.class, () -> {
                r5.setName("Milk");
                r5.setAmtChocolate("aswd");
                r5.setAmtCoffee("fgd");
                r5.setAmtMilk("4");
                r5.setAmtSugar(null);
                r5.setPrice("9999999999999999999999999");
                cm.addRecipe(r5);
            }); // Should throw an Exception
}
```

5



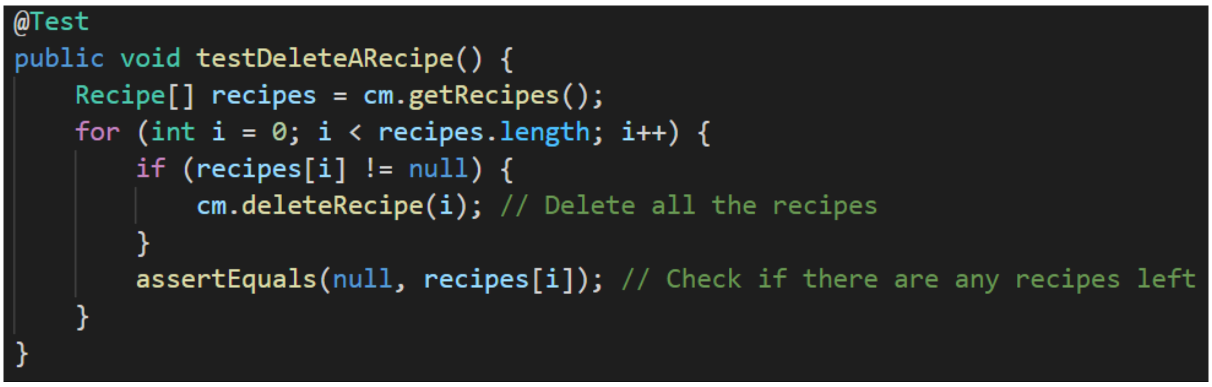
```
@Test
public void testEditRecipe() {
    Recipe[] recipes = cm.getRecipes();
    Recipe newRecipe = new Recipe();
    String recipeEdited = null;
    try {
        newRecipe.setName("Milk");
        newRecipe.setPrice("20");
        newRecipe.setAmtCoffee("1");
        newRecipe.setAmtMilk("4");
        newRecipe.setAmtSugar("2");
        newRecipe.setAmtChocolate("0");
        recipeEdited = cm.editRecipe(0, newRecipe); // We try to edit the first recepi
    } catch (RecipeException e) {
        fail("RecipeException should not be throun.");
    }
    assertEquals(recipeEdited, recipes[0]); // Check if they are the same
}
```

6



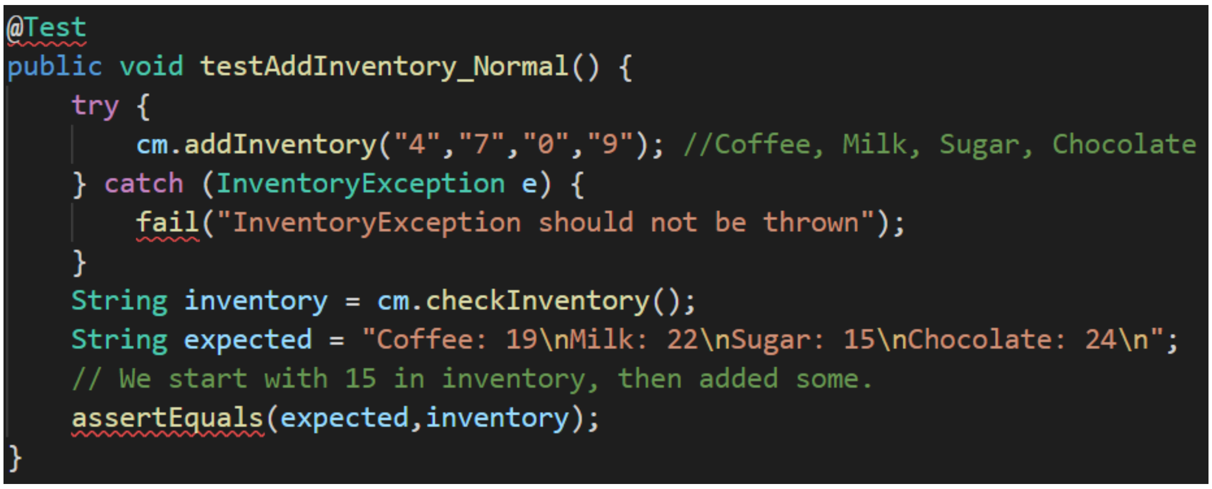
```java
@Test
public void testEditRecipeException() {
    Recipe[] recipes = cm.getRecipes();
    Recipe newRecipe = new Recipe();
    Throwable exception = assertThrows(
            RecipeException.class, () -> {
                newRecipe.setName("123");
                newRecipe.setPrice(null);
                newRecipe.setAmtCoffee("a");
                newRecipe.setAmtMilk("b");
                newRecipe.setAmtSugar("2");
                newRecipe.setAmtChocolate("x");
                cm.editRecipe(0, newRecipe);
            }); // Should throw an Exception
}
```

7



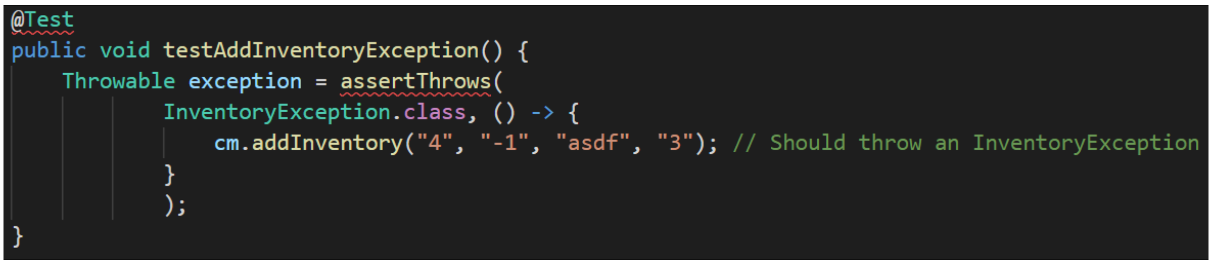
```java
@Test
public void testDeleteARecipe() {
    Recipe[] recipes = cm.getRecipes();
    for (int i = 0; i < recipes.length; i++) {
        if (recipes[i] != null) {
            cm.deleteRecipe(i); // Delete all the recipes
        }
        assertEquals(null, recipes[i]); // Check if there are any recipes left
    }
}
```

8



```java
@Test
public void testAddInventory_Normal() {
    try {
        cm.addInventory("4","7","0","9"); //Coffee, Milk, Sugar, Chocolate
    } catch (InventoryException e) {
        fail("InventoryException should not be thrown");
    }
    String inventory = cm.checkInventory();
    String expected = "Coffee: 19\nMilk: 22\nSugar: 15\nChocolate: 24\n";
    // We start with 15 in inventory, then added some.
    assertEquals(expected,inventory);
}
```

9



```java
@Test
public void testAddInventoryException() {
    Throwable exception = assertThrows(
            InventoryException.class, () -> {
                cm.addInventory("4", "-1", "asdf", "3"); // Should throw an InventoryException
            }
            );
}
```

3. Instructions on how to set up and execute your tests (if you used any external libraries other than JUnit itself, or did anything non-obvious when creating your unit tests. You may include a build script if you created one).

Running the tests should be as easy as navigating to the CoffeeMaker folder and running the command “**mvn test**” after you have installed Maven.

In case it does not run for all files or you want to run a specific test class, try:
mvn clean test -Dtest=NameOfTheTestClass

4. List of faults found, along with a fix for each and a list of which of your test cases expose the fault.
   First, we had to fix some bugs.

   1. When making coffee, the coffee value would increase instead of decreasing. Our fix is:
      CoffeeMaker\src\main\java\edu\ncsu\csc326\coffeemaker\Inventory.java

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

   2. When adding ingredients, sugar would behave weirdly.
      CoffeeMaker\src\main\java\edu\ncsu\csc326\coffeemaker\Inventory.java

```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");
        }
    }
```

3. When editing a recipe, the name would disappear.
CoffeeMaker\src\main\java\edu\ncsu\csc326\coffeemaker\Main.java

```java
        //Read recipe name
        String nameString = inputOutput("\nPlease enter the recipe
name: ");

        //Read in recipe price
        String priceString = inputOutput("\nPlease enter the
recipe price: $");

        //Read in amt coffee
        String coffeeString = inputOutput("\nPlease enter the
units of coffee in the recipe: ");

        //Read in amt milk
        String milkString = inputOutput("\nPlease enter the units
of milk in the recipe: ");

        //Read in amt sugar
        String sugarString = inputOutput("\nPlease enter the units
of sugar in the recipe: ");

        //Read in amt chocolate
        String chocolateString = inputOutput("\nPlease enter the
units of chocolate in the recipe: ");

        Recipe newRecipe = new Recipe();
        try {
            newRecipe.setName(nameString);
            newRecipe.setPrice(priceString);
            newRecipe.setAmtCoffee(coffeeString);
            newRecipe.setAmtMilk(milkString);
            newRecipe.setAmtSugar(sugarString);
            newRecipe.setAmtChocolate(chocolateString);

            String recipeEdited =
coffeeMaker.editRecipe(recipeToEdit, newRecipe);
```

CoffeeMaker\src\main\java\edu\ncsu\csc326\coffeemaker\Recipe.java

```java
    public String getName() {
        return this.name;
    }
```

CoffeeMaker\src\main\java\edu\ncsu\csc326\coffeemaker\RecipeBook.java

```java
    public synchronized String editRecipe(int recipeToEdit, Recipe
newRecipe) {
        if (recipeArray[recipeToEdit] != null) {
            recipeArray[recipeToEdit] = newRecipe;
            return recipeArray[recipeToEdit].getName();
        } else {
            return null;
        }
    }
```

Problem 2 - Structural Coverage

CoffeeMaker::makeCoffee

testMakeCoffee_null_recipe tests the case in which the recipe to purchase is null, this triggers the first outer if-statement on line 89. The user gets their money refunded

testMakeCoffee_normal tests the case in which every input is normal, meaning that the recipe exists, the user has enough money and there is enough inventory. This triggers the second outer if-statement on line 91 and the inner if-statement on line 92. If the user has paid more than enough, they get their change.

testMakeCoffee_not_enough_inventory is very similar to the previous test, the difference is that there are not enough materials (inventory) to make the coffee. This triggers the inner else statement on line 94 and the user gets their money refunded.

testMakeCoffee_not_enough_money covers the case in which the user does not have enough money to purchase the desired drink, this triggers the outer else statement on line 97 and the user gets their money refunded.

Inventory::addSugar

testAddSugar_normal tests the normal functionality of the “addSugar” function by adding 10 pieces of sugar to the system and testing it to make sure that the correct number of sugar is present (lines 176, 178, 182, 183).

testAddSugar_String will try to add a string instead of a number, triggering the “InventoryException” (lines 179, 180).

*testAddSugar_neg_numbe*r will try to add a negative number of sugar to the system. This will trigger the second “InventoryException” of the function (line 185).

Recipe::setPrice

We have three tests for the setPrice function.

testSetPrice_normal tests the functionality of the “setPrice” function by setting the price for the given recipe to 10 in the system and testing it to make sure the correct price is shown in the system (lines 140, 142, 147).

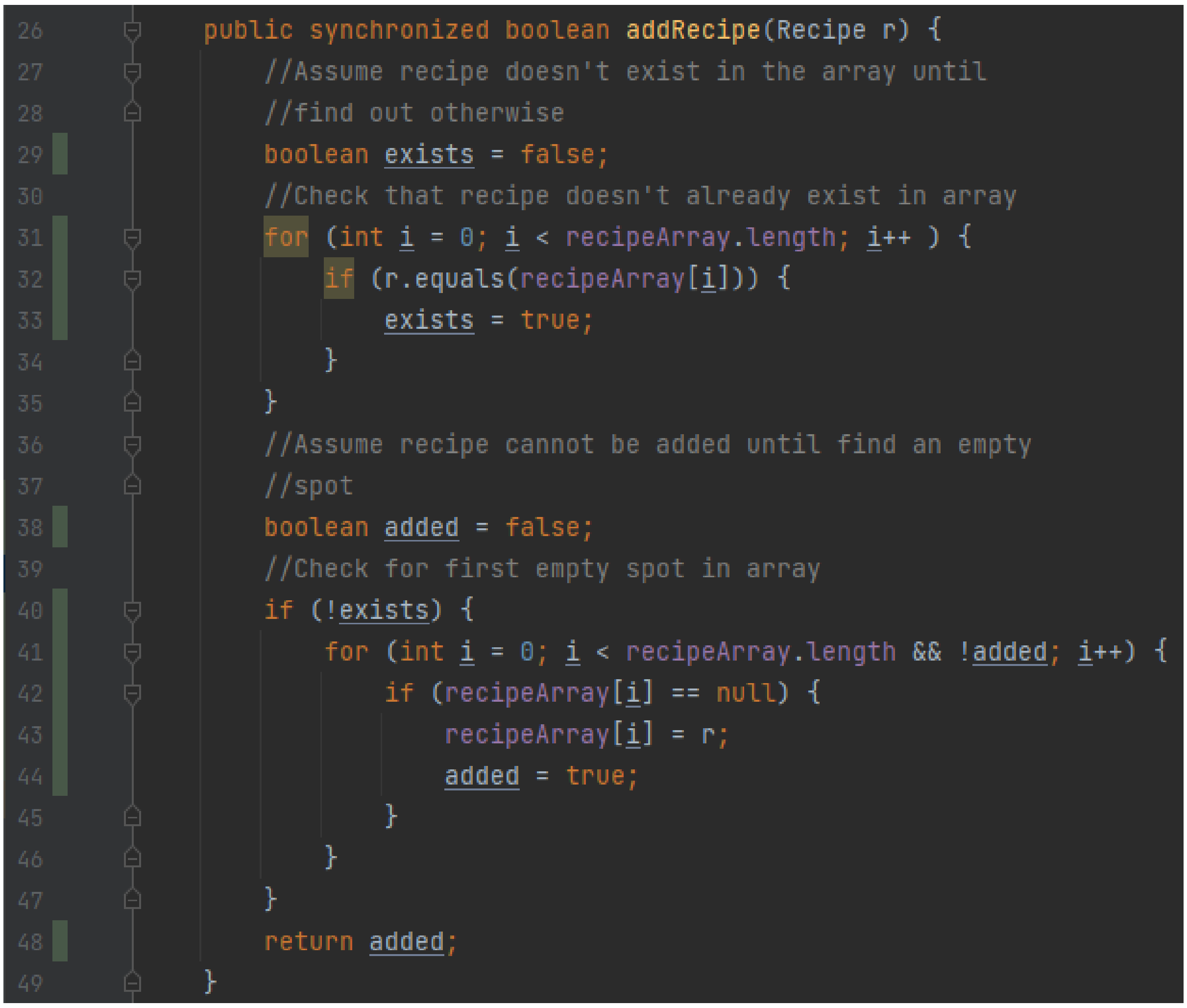
testSetPrice_String this test tests by trying to run a string in place of a number, which will trigger the “RecipeException” of the function (lines 143, 144).

testSetPrice_neg_number this test will try to add and run a negative number for the price of the drink in the system. This would trigger the “RecipeException” of the function (line 149).

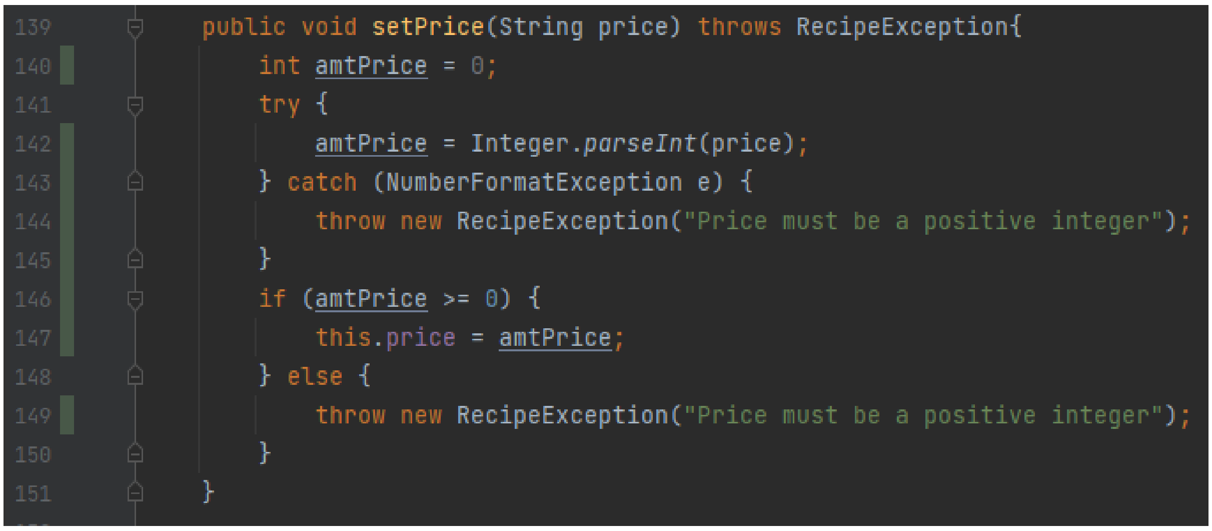
RecipeBook::addRecipe

Firstly we create a “RecipeBook” and a “Recipe” object. We will then add it to the system, covering the second half of the “addRecipe” function (from line 40). Then, we will add the same recipe again, triggering the coverage on the first half of the function by setting “exists” to true (from line 31).

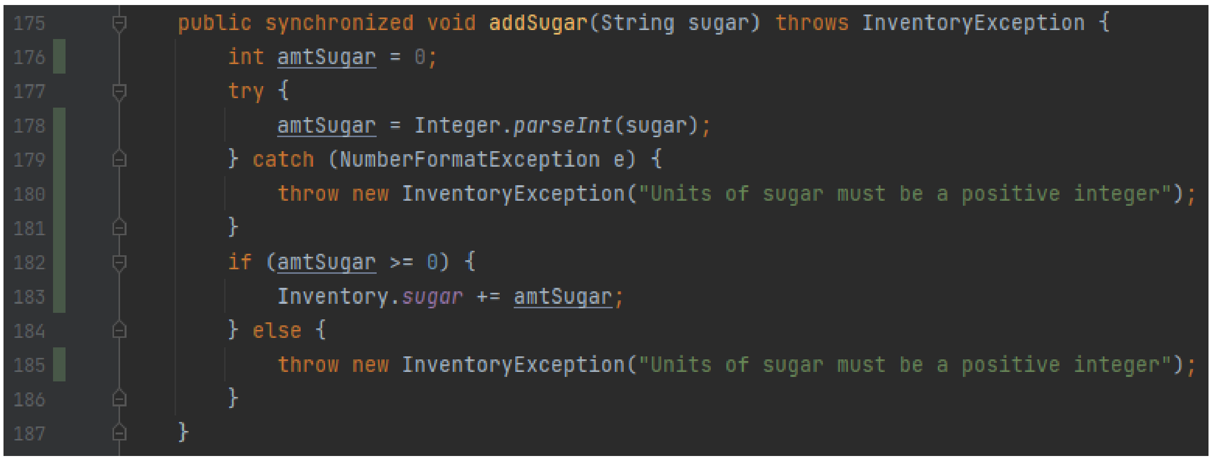
2. Green lines on the left-side indicate lines that are covered.



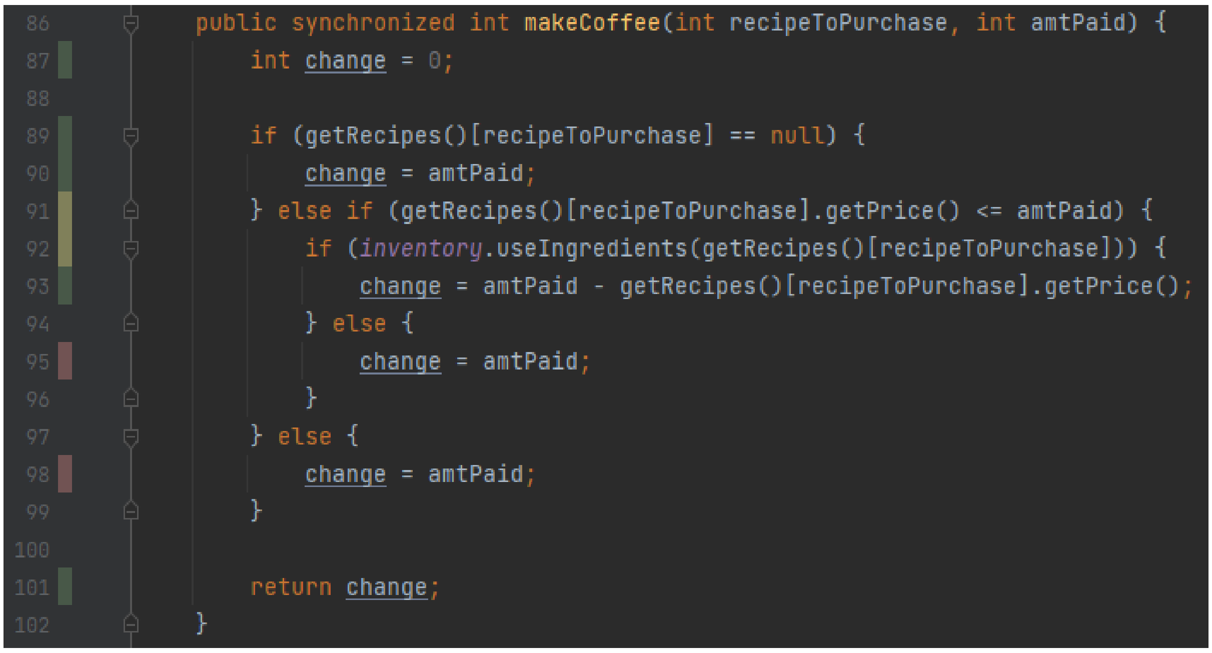
```
26      public synchronized boolean addRecipe(Recipe r) {
27          //Assume recipe doesn't exist in the array until
28          //find out otherwise
29          boolean exists = false;
30          //Check that recipe doesn't already exist in array
31          for (int i = 0; i < recipeArray.length; i++ ) {
32              if (r.equals(recipeArray[i])) {
33                  exists = true;
34              }
35          }
36          //Assume recipe cannot be added until find an empty
37          //spot
38          boolean added = false;
39          //Check for first empty spot in array
40          if (!exists) {
41              for (int i = 0; i < recipeArray.length && !added; i++) {
42                  if (recipeArray[i] == null) {
43                      recipeArray[i] = r;
44                      added = true;
45                  }
46              }
47          }
48          return added;
49      }
```



```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");
        }
    }
```



```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");
        }
    }
```



```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 you can see, we achieve 100% Line Coverage in every case except for CoffeeMaker::makeCoffee. We spent a lot of effort on finding out why but we could not come up with a solution in the end. In our opinion, the tests should be fine as they cover all of the branches which should therefore trigger all the lines.