CS 583, Assignment 3

Authors: William Rosenberger, Christina Yu

Introduction

The following document describes the test cases that were built for the CoffeeMaker project. We have concentrated on testing the individual use cases in this project rather than the specific classes making it up. That is, our focus was on testing the interactions between the classes in the system.

Note that there was a fault during the initial design of this piece of software. There is not a clean break between the user interface and the business logic aspect of the package. In other words, user input, output, error checking, and data processing are frequently mixed together in the same methods. This is an issue for testing as it makes it significantly more difficult to inject testing data and to intercept messages between systems to ensure they contain the expected responses. To increase the lifetime and agility of the software, the test team recommends redesigning the user interface with the Model-View-Controller pattern in mind. Due to this design issue, we chose to dedicate our resources to testing those sections of the code base over which we have a high degree of control.

Use Case: Add a Recipe

The "Add a Recipe" use case is a high-priority task for the CoffeeMaker software system. The following test cases were created to cover this use case:

1. Check that a recipe can be added when no recipes currently exist.

2. Check that more than one recipe can be added.

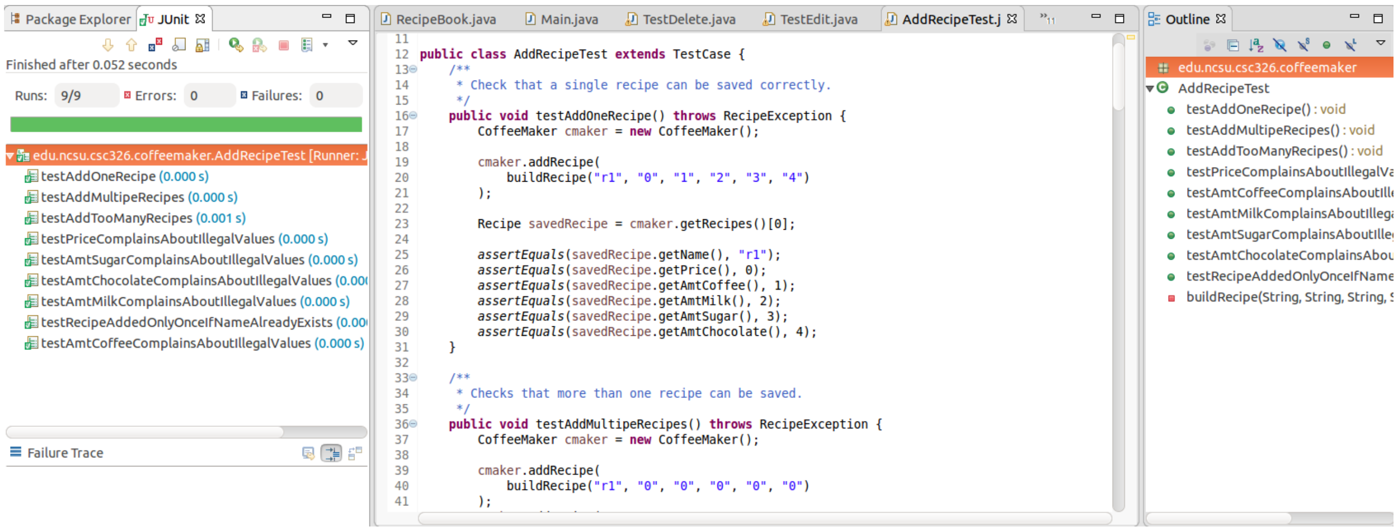
3. Check that the correct error is thrown if too many recipes are added (the system is designed to accept a maximum of 3 recipes).

   Bug discovered: The system was not checking this requirement. This would throw and unhandled exception if the user attempted to add more than 3 recipes. This bug has been fixed.

4. Check that the correct errors are thrown if any of the properties are given NaN values. For example, we ensure that an error is raised if an attempt is made to create a coffee recipe with "dog" units of chocolate.

5. Check that the correct errors are thrown if any of the properties are given negative values.

6. Check that a new recipe is *not* created if there is already a recipe with that name.



```
Finished after 0.052 seconds
Runs: 9/9    Errors: 0    Failures: 0

edu.ncsu.csc326.coffeemaker.AddRecipeTest [Runner: J
    testAddOneRecipe (0.000 s)
    testAddMultipeRecipes (0.000 s)
    testAddTooManyRecipes (0.001 s)
    testPriceComplainsAboutIllegalValues (0.000 s)
    testAmtSugarComplainsAboutIllegalValues (0.000 s)
    testAmtChocolateComplainsAboutIllegalValues (0.00
    testAmtMilkComplainsAboutIllegalValues (0.000 s)
    testRecipeAddedOnlyOnceIfNameAlreadyExists (0.00
    testAmtCoffeeComplainsAboutIllegalValues (0.000 s)
```

```java

public class AddRecipeTest extends TestCase {
    /**
     * Check that a single recipe can be saved correctly.
     */
    public void testAddOneRecipe() throws RecipeException {
        CoffeeMaker cmaker = new CoffeeMaker();

        cmaker.addRecipe(
            buildRecipe("r1", "0", "1", "2", "3", "4")
        );

        Recipe savedRecipe = cmaker.getRecipes()[0];

        assertEquals(savedRecipe.getName(), "r1");
        assertEquals(savedRecipe.getPrice(), 0);
        assertEquals(savedRecipe.getAmtCoffee(), 1);
        assertEquals(savedRecipe.getAmtMilk(), 2);
        assertEquals(savedRecipe.getAmtSugar(), 3);
        assertEquals(savedRecipe.getAmtChocolate(), 4);
    }

    /**
     * Checks that more than one recipe can be saved.
     */
    public void testAddMultipeRecipes() throws RecipeException {
        CoffeeMaker cmaker = new CoffeeMaker();

        cmaker.addRecipe(
            buildRecipe("r1", "0", "0", "0", "0", "0")
        );
```

```
edu.ncsu.csc326.coffeemaker
AddRecipeTest
    testAddOneRecipe() : void
    testAddMultipeRecipes() : void
    testAddTooManyRecipes() : void
    testPriceComplainsAboutIllegalVa
    testAmtCoffeeComplainsAboutIll
    testAmtMilkComplainsAboutIllega
    testAmtSugarComplainsAboutIlle
    testAmtChocolateComplainsAbou
    testRecipeAddedOnlyOnceIfName
    buildRecipe(String, String, String, S
```

Use Case: Delete a Recipe

The "Delete a Recipe" use case was identified as a low-priority task. The following test cases were created to cover this use case:

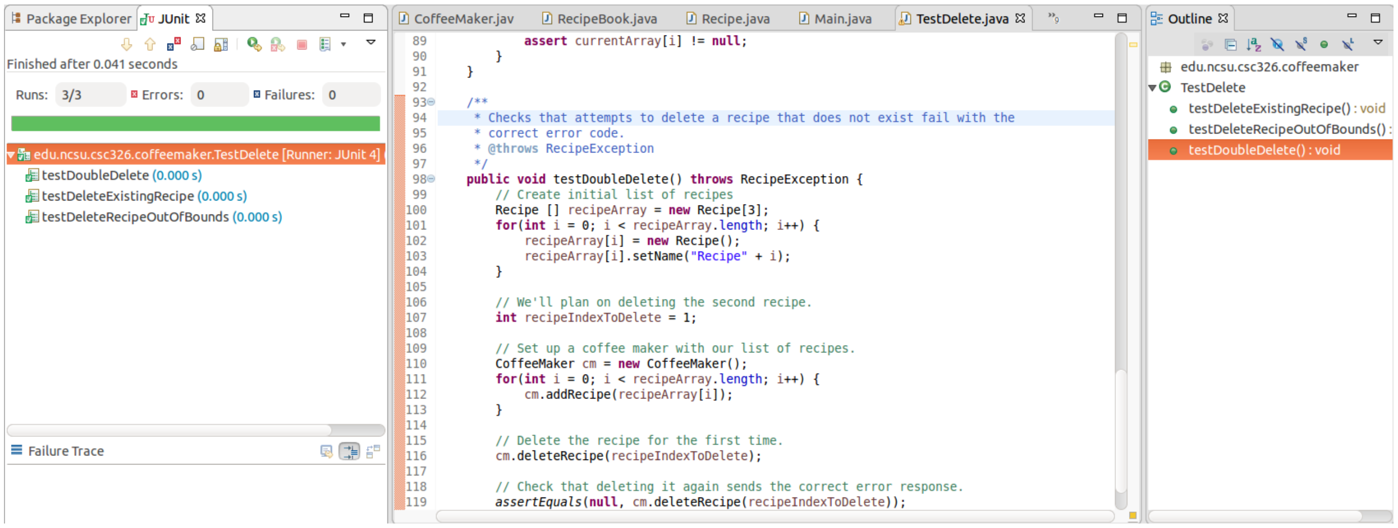
1. Check that deleting a recipe that exists correctly removes the recipe from the coffee maker.

   Bug discovered: Rather than deleting the recipe, it was merely overwritten with a new, blank recipe. This made it look like the recipe book was still full, even if a recipe was deleted. This was corrected by setting the deleted recipe to `null` to indicate that the slot in the recipe book is open.

2. Check that requesting the deletion of a recipe that is outside the bounds of the recipe array fails with the correct error signal.

   Bug discovered: No checks were made within the `RecipeBook` implementation itself to ensure the requested index is within the array bounds. This check was only implemented within the user interface. However, `RecipeBook.deleteRecipe()` is a public method, which means any system is allowed to call it. This means that a new system may be added down the line that calls `RecipeBook.deleteRecipe()` but does not perform the necessary range checks. For this reason, it is important to check the input range in `RecipeBook.deleteRecipe()` itself. We have added this check.

3. Check that attempts to delete a recipe that does not exist fail with the expected error code.



Finished after 0.041 seconds

Runs: 3/3 Errors: 0 Failures: 0

edu.ncsu.csc326.coffeemaker.TestDelete [Runner: JUnit 4]
testDoubleDelete (0.000 s)
testDeleteExistingRecipe (0.000 s)
testDeleteRecipeOutOfBounds (0.000 s)

```java
            assert currentArray[i] != null;
        }
    }

    /**
     * Checks that attempts to delete a recipe that does not exist fail with the
     * correct error code.
     * @throws RecipeException
     */
    public void testDoubleDelete() throws RecipeException {
        // Create initial list of recipes
        Recipe [] recipeArray = new Recipe[3];
        for(int i = 0; i < recipeArray.length; i++) {
            recipeArray[i] = new Recipe();
            recipeArray[i].setName("Recipe" + i);
        }

        // We'll plan on deleting the second recipe.
        int recipeIndexToDelete = 1;

        // Set up a coffee maker with our list of recipes.
        CoffeeMaker cm = new CoffeeMaker();
        for(int i = 0; i < recipeArray.length; i++) {
            cm.addRecipe(recipeArray[i]);
        }

        // Delete the recipe for the first time.
        cm.deleteRecipe(recipeIndexToDelete);

        // Check that deleting it again sends the correct error response.
        assertEquals(null, cm.deleteRecipe(recipeIndexToDelete));
```

edu.ncsu.csc326.coffeemaker
TestDelete
testDeleteExistingRecipe() : void
testDeleteRecipeOutOfBounds() :
testDoubleDelete() : void

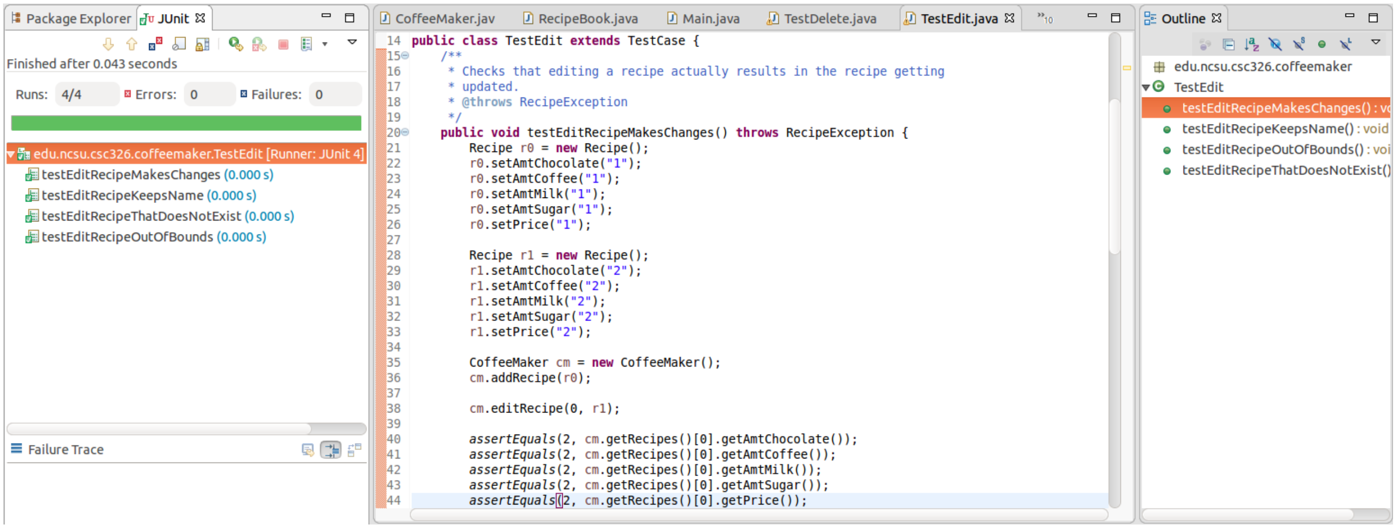
Use Case: Edit a Recipe

The "Edit a Recipe" use case was identified as a low-priority task. The following test cases were created to cover this use case:

1. Check that all properties of a recipe (e.g., price, amount of coffee) are updated.
2. Check that the name of a recipe is kept constant between edits

   Bug discovered: Initial code would set the edited recipe's name to the empty string, rather than to the original string. This has been fixed.

3. Check that `RecipeBook` 's `editRecipe()` method checks that the requested index is within the array's range. Previously, this was only checked within the user interface. However, as was the case with the `deleteRecipe()` method, it is important that this condition is checked within the `RecipeBook` class itself (public methods may be called from anywhere, and it is therefore unsafe to assume the correct input checks are in place).
4. Checks that the correct response is given if the user tries to edit a recipe that does not exist.



```
Finished after 0.043 seconds
Runs: 4/4    Errors: 0    Failures: 0

edu.ncsu.csc326.coffeemaker.TestEdit [Runner: JUnit 4]
    testEditRecipeMakesChanges (0.000 s)
    testEditRecipeKeepsName (0.000 s)
    testEditRecipeThatDoesNotExist (0.000 s)
    testEditRecipeOutOfBounds (0.000 s)
```

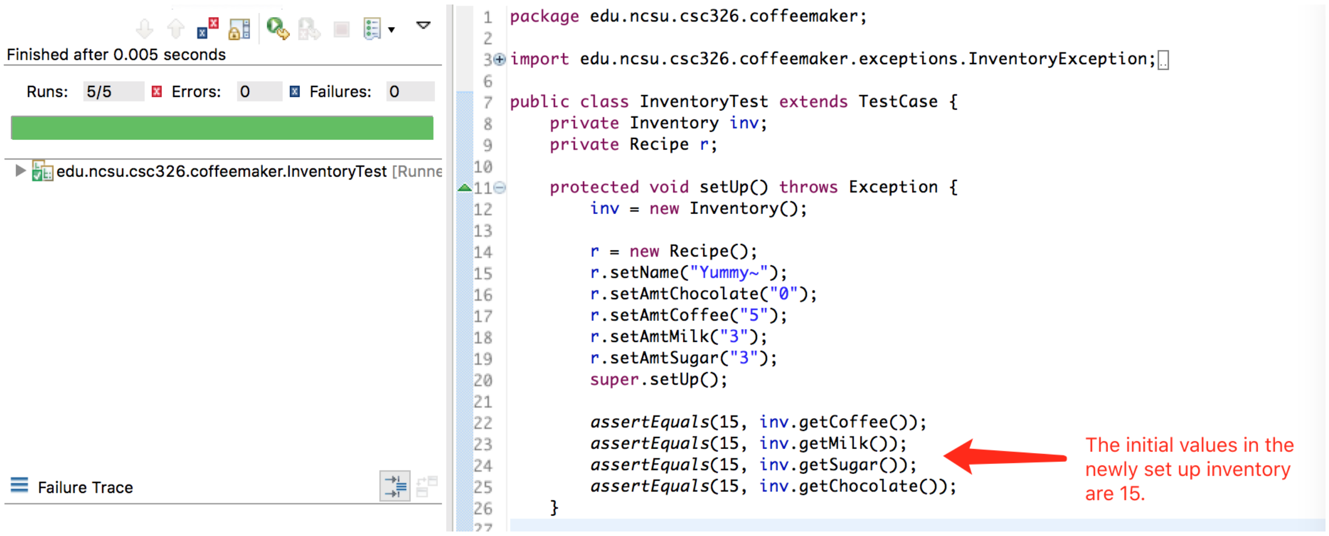
```java
public class TestEdit extends TestCase {
    /**
     * Checks that editing a recipe actually results in the recipe getting
     * updated.
     * @throws RecipeException
     */
    public void testEditRecipeMakesChanges() throws RecipeException {
        Recipe r0 = new Recipe();
        r0.setAmtChocolate("1");
        r0.setAmtCoffee("1");
        r0.setAmtMilk("1");
        r0.setAmtSugar("1");
        r0.setPrice("1");

        Recipe r1 = new Recipe();
        r1.setAmtChocolate("2");
        r1.setAmtCoffee("2");
        r1.setAmtMilk("2");
        r1.setAmtSugar("2");
        r1.setPrice("2");

        CoffeeMaker cm = new CoffeeMaker();
        cm.addRecipe(r0);

        cm.editRecipe(0, r1);

        assertEquals(2, cm.getRecipes()[0].getAmtChocolate());
        assertEquals(2, cm.getRecipes()[0].getAmtCoffee());
        assertEquals(2, cm.getRecipes()[0].getAmtMilk());
        assertEquals(2, cm.getRecipes()[0].getAmtSugar());
        assertEquals(2, cm.getRecipes()[0].getPrice());
```

Package Explorer
JUnit
Failure Trace
CoffeeMaker.jav
RecipeBook.java
Main.java
TestDelete.java
TestEdit.java
Outline
edu.ncsu.csc326.coffeemaker
TestEdit
testEditRecipeMakesChanges() : v
testEditRecipeKeepsName() : void
testEditRecipeOutOfBounds() : voi
testEditRecipeThatDoesNotExist()

Use Case: Add Inventory

The "Add Inventory" use case is a high-priority task for the CoffeeMaker software system. The following test cases were created to cover this use case:

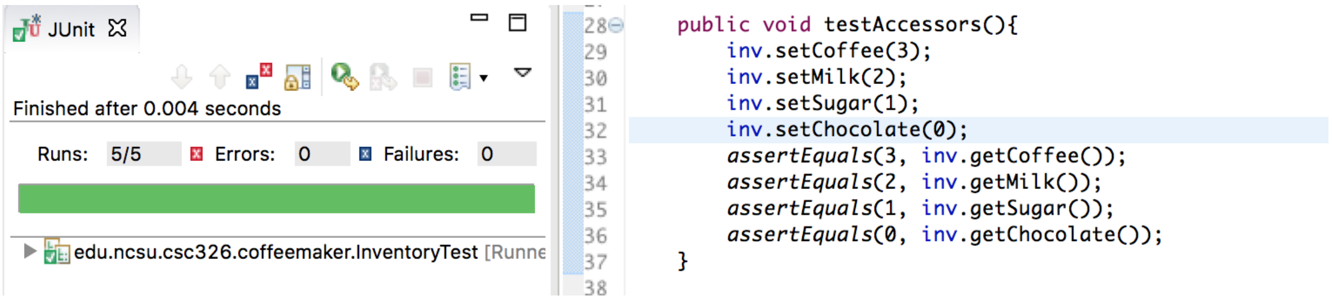
1. Check that an inventory can be set up with correct initial values.



```
Finished after 0.005 seconds
Runs: 5/5    Errors: 0    Failures: 0

edu.ncsu.csc326.coffeemaker.InventoryTest [Runne
```

```java
package edu.ncsu.csc326.coffeemaker;

import edu.ncsu.csc326.coffeemaker.exceptions.InventoryException;

public class InventoryTest extends TestCase {
    private Inventory inv;
    private Recipe r;

    protected void setUp() throws Exception {
        inv = new Inventory();

        r = new Recipe();
        r.setName("Yummy~");
        r.setAmtChocolate("0");
        r.setAmtCoffee("5");
        r.setAmtMilk("3");
        r.setAmtSugar("3");
        super.setUp();

        assertEquals(15, inv.getCoffee());
        assertEquals(15, inv.getMilk());
        assertEquals(15, inv.getSugar());
        assertEquals(15, inv.getChocolate());
    }
```

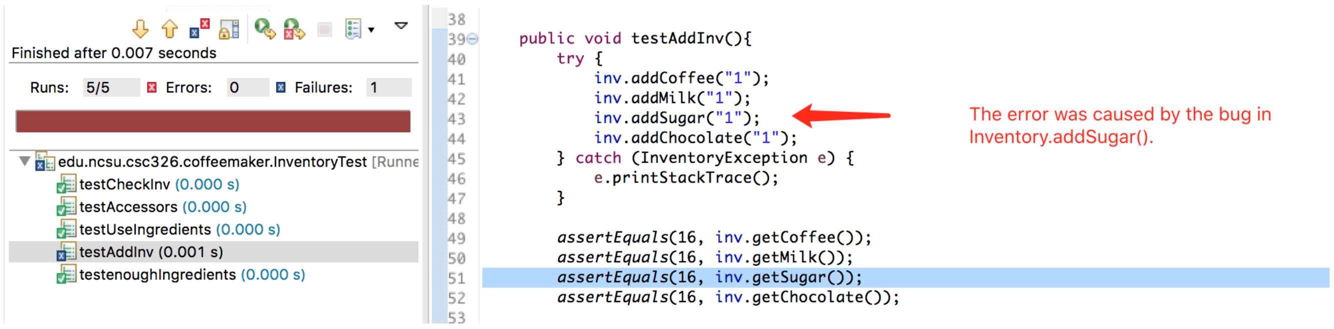
2. Check that the accessors(get() and set() methods) are setting and getting the correct value.



```
JUnit
Finished after 0.004 seconds
Runs: 5/5    Errors: 0    Failures: 0

edu.ncsu.csc326.coffeemaker.InventoryTest [Runne
```

```java
    public void testAccessors(){
        inv.setCoffee(3);
        inv.setMilk(2);
        inv.setSugar(1);
        inv.setChocolate(0);
        assertEquals(3, inv.getCoffee());
        assertEquals(2, inv.getMilk());
        assertEquals(1, inv.getSugar());
        assertEquals(0, inv.getChocolate());
    }
```

3. Check that unit(s) of Coffee, Milk, Sugar and Chocolate can be added to an existing inventory.


Finished after 0.007 seconds

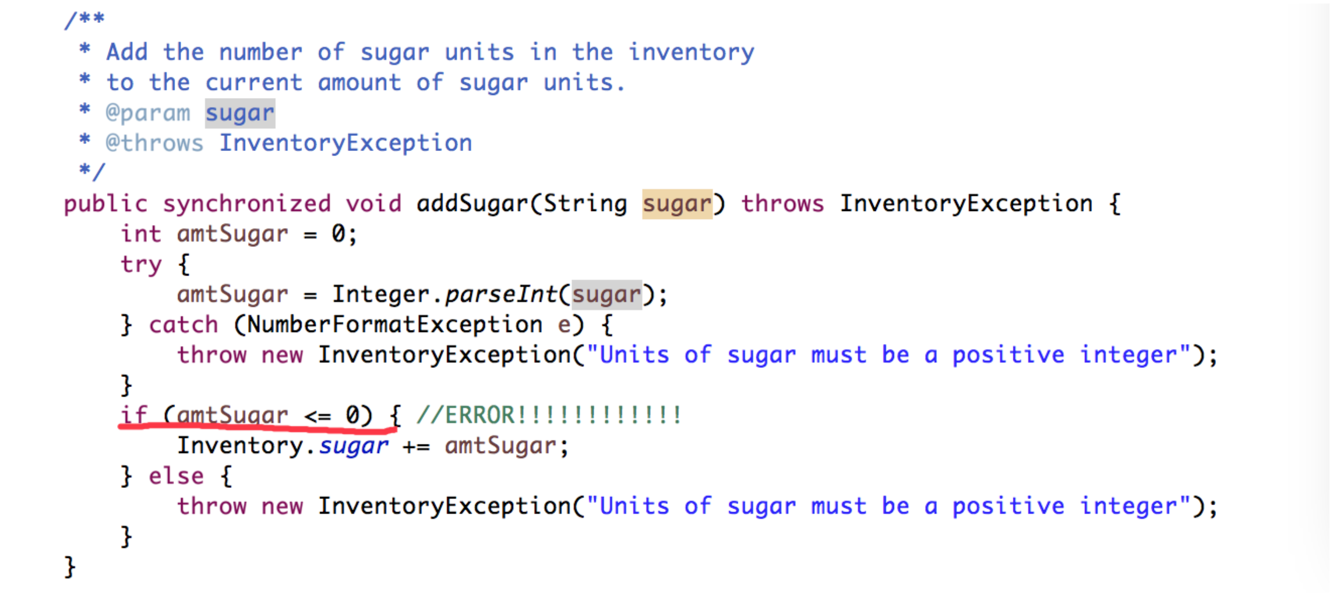
Runs: 5/5 Errors: 0 Failures: 1

edu.ncsu.csc326.coffeemaker.InventoryTest [Runne
testCheckInv (0.000 s)
testAccessors (0.000 s)
testUseIngredients (0.000 s)
testAddInv (0.001 s)
testenoughIngredients (0.000 s)

```
public void testAddInv(){
    try {
        inv.addCoffee("1");
        inv.addMilk("1");
        inv.addSugar("1");
        inv.addChocolate("1");
    } catch (InventoryException e) {
        e.printStackTrace();
    }

    assertEquals(16, inv.getCoffee());
    assertEquals(16, inv.getMilk());
    assertEquals(16, inv.getSugar());
    assertEquals(16, inv.getChocolate());
```

The error was caused by the bug in Inventory.addSugar().

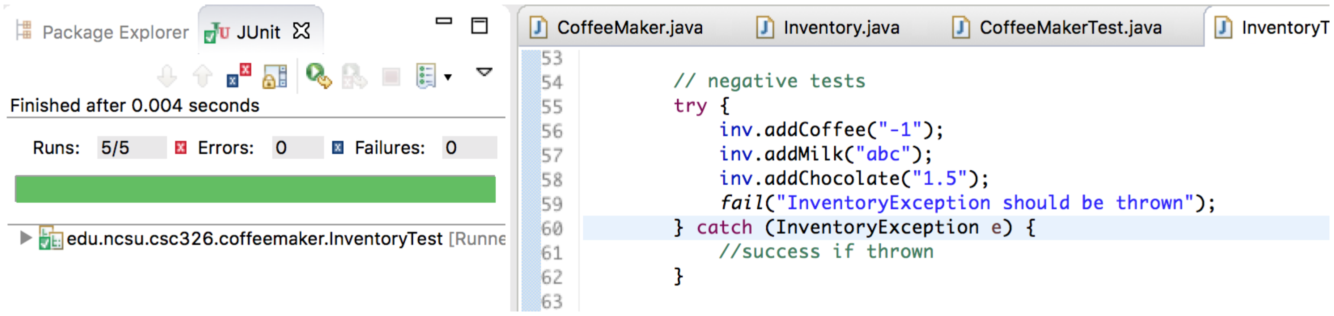
Bug discovered: The bug was found in Inventory.addSugar(). The "amtSugar" should be a positive number or zero, instead of "amtSugar <= 0". This bug has been fixed.


```
/**
 * Add the number of sugar units in the inventory
 * to the current amount of sugar units.
 * @param sugar
 * @throws InventoryException
 */
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) { //ERROR!!!!!!!!!!!!
        Inventory.sugar += amtSugar;
    } else {
        throw new InventoryException("Units of sugar must be a positive integer");
    }
}
```

4. Check that the correct errors are thrown if the wrong form of the ingredient was added to the inventory. For example, if "abc" unit of Milk is added to the inventory, the exception should be thrown.

5. Check that the correct errors are thrown if any of the properties are given negative values.

6. Check that the correct errors are thrown if a non-integer number unit is added to the inventory.



Finished after 0.004 seconds

Runs: 5/5 Errors: 0 Failures: 0

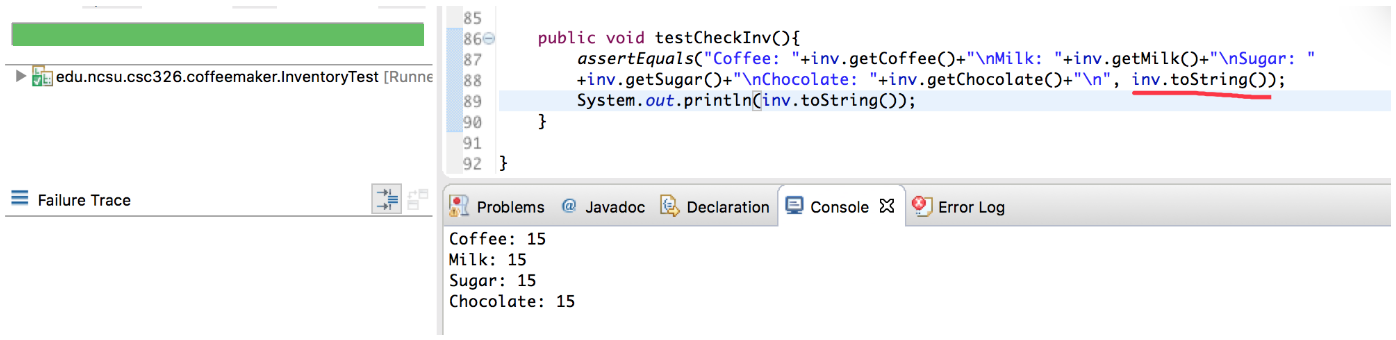
edu.ncsu.csc326.coffeemaker.InventoryTest [Runne

```
        // negative tests
        try {
            inv.addCoffee("-1");
            inv.addMilk("abc");
            inv.addChocolate("1.5");
            fail("InventoryException should be thrown");
        } catch (InventoryException e) {
            //success if thrown
        }
```

Use Case: Check Inventory

The "Check Inventory" use case was identified as a low-priority task. The following test case was created to cover this use case:

Check that if the units of each ingredient in the inventory are displayed on the list with correct values and format.



edu.ncsu.csc326.coffeemaker.InventoryTest [Runne

Failure Trace

```java
    public void testCheckInv(){
        assertEquals("Coffee: "+inv.getCoffee()+"\nMilk: "+inv.getMilk()+"\nSugar: "
        +inv.getSugar()+"\nChocolate: "+inv.getChocolate()+"\n", inv.toString());
        System.out.println(inv.toString());
    }

}
```

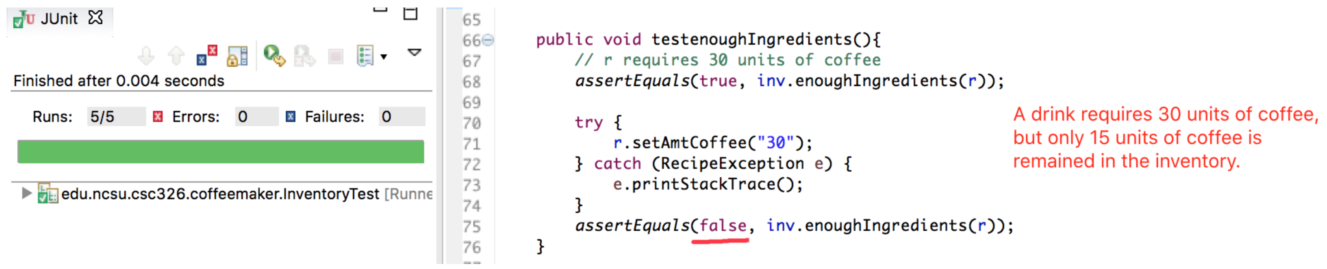
Problems | Javadoc | Declaration | Console | Error Log

```
Coffee: 15
Milk: 15
Sugar: 15
Chocolate: 15
```

Use Case: Purchase Coffee

The "Purchase Coffee" use case is a high-priority task for the CoffeeMaker software system. The following test cases were created to cover this use case:

1. Test the function of checking if there are enough ingredients in the inventory to make the selected drink.



JUnit

Finished after 0.004 seconds

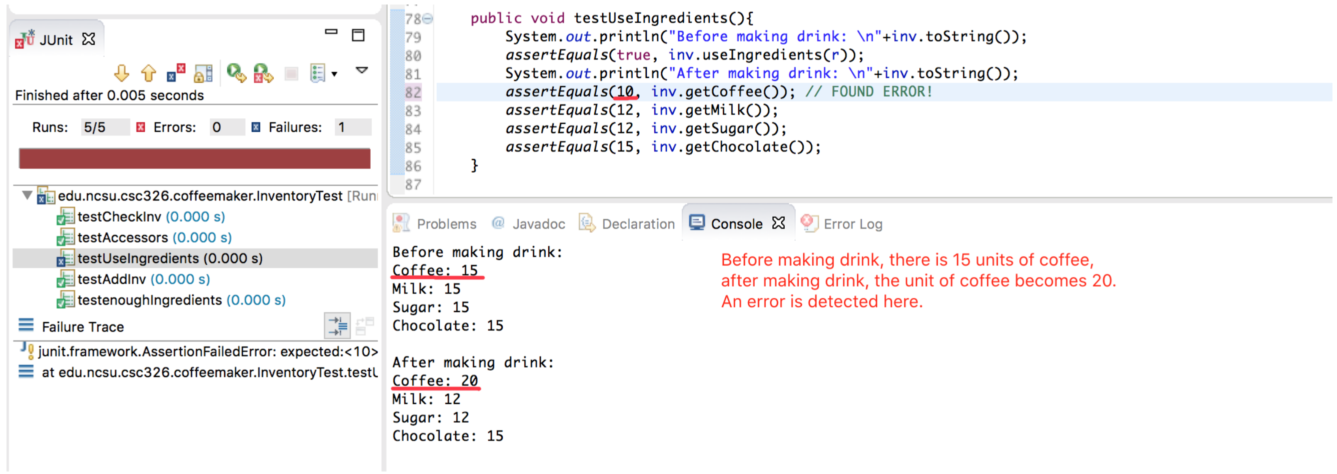
Runs: 5/5 Errors: 0 Failures: 0

edu.ncsu.csc326.coffeemaker.InventoryTest [Runne

```java
    public void testenoughIngredients(){
        // r requires 30 units of coffee
        assertEquals(true, inv.enoughIngredients(r));

        try {
            r.setAmtCoffee("30");
        } catch (RecipeException e) {
            e.printStackTrace();
        }
        assertEquals(false, inv.enoughIngredients(r));
    }
```

A drink requires 30 units of coffee, but only 15 units of coffee is remained in the inventory.

2. Check that the amount of ingredient is subtracted from the inventory when making a drink.



JUnit

Finished after 0.005 seconds

Runs: 5/5 Errors: 0 Failures: 1

edu.ncsu.csc326.coffeemaker.InventoryTest [Run
testCheckInv (0.000 s)
testAccessors (0.000 s)
testUseIngredients (0.000 s)
testAddInv (0.000 s)
testenoughIngredients (0.000 s)

Failure Trace

```
junit.framework.AssertionFailedError: expected:<10>
at edu.ncsu.csc326.coffeemaker.InventoryTest.testL
```

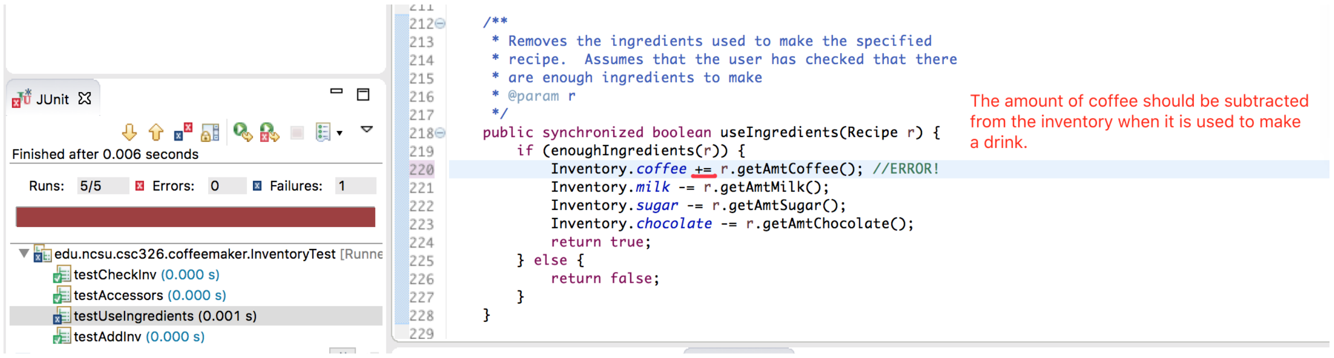
```java
    public void testUseIngredients(){
        System.out.println("Before making drink: \n"+inv.toString());
        assertEquals(true, inv.useIngredients(r));
        System.out.println("After making drink: \n"+inv.toString());
        assertEquals(10, inv.getCoffee()); // FOUND ERROR!
        assertEquals(12, inv.getMilk());
        assertEquals(12, inv.getSugar());
        assertEquals(15, inv.getChocolate());
    }
```

Problems | Javadoc | Declaration | Console | Error Log

```
Before making drink:
Coffee: 15
Milk: 15
Sugar: 15
Chocolate: 15

After making drink:
Coffee: 20
Milk: 12
Sugar: 12
Chocolate: 15
```

Before making drink, there is 15 units of coffee, after making drink, the unit of coffee becomes 20. An error is detected here.

Bug discovered: The bug was found in Inventory.useIngredients(Recipe r). The amount of coffee a recipe requires should be subtracted from the inventory when making a drink, instead of adding the amount of coffee to inventory. This bug has been fixed.



JUnit

Finished after 0.006 seconds

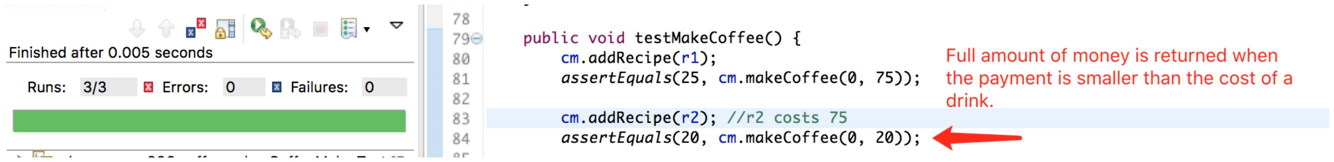
Runs: 5/5 Errors: 0 Failures: 1

- edu.ncsu.csc326.coffeemaker.InventoryTest [Runne
  - testCheckInv (0.000 s)
  - testAccessors (0.000 s)
  - testUseIngredients (0.001 s)
  - testAddInv (0.000 s)

```
    /**
     * Removes the ingredients used to make the specified
     * recipe.  Assumes that the user has checked that there
     * are enough ingredients to make
     * @param r
     */
    public synchronized boolean useIngredients(Recipe r) {
        if (enoughIngredients(r)) {
            Inventory.coffee += r.getAmtCoffee(); //ERROR!
            Inventory.milk -= r.getAmtMilk();
            Inventory.sugar -= r.getAmtSugar();
            Inventory.chocolate -= r.getAmtChocolate();
            return true;
        } else {
            return false;
        }
    }
```

The amount of coffee should be subtracted from the inventory when it is used to make a drink.

3. Check that if the user does not enter enough money, their money will be returned.



Finished after 0.005 seconds

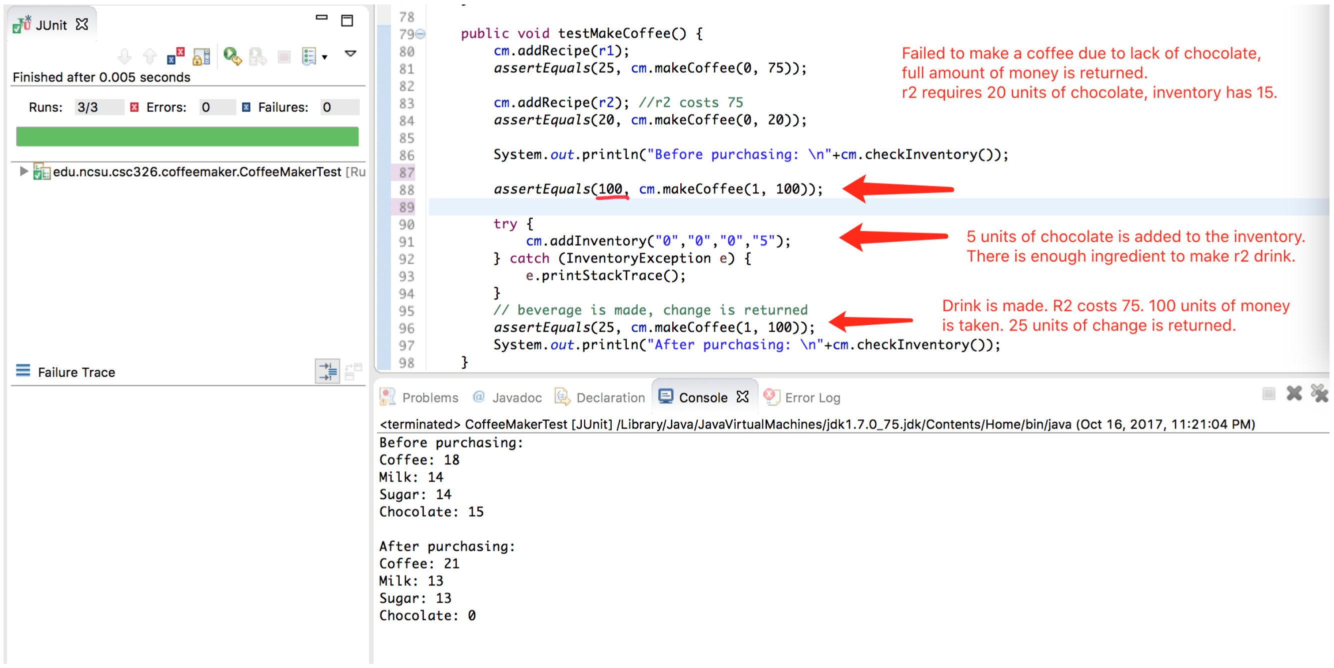
Runs: 3/3 Errors: 0 Failures: 0

```
    public void testMakeCoffee() {
        cm.addRecipe(r1);
        assertEquals(25, cm.makeCoffee(0, 75));

        cm.addRecipe(r2); //r2 costs 75
        assertEquals(20, cm.makeCoffee(0, 20));
```

Full amount of money is returned when the payment is smaller than the cost of a drink.

4. Check that if there is not enough inventory to make the beverage, the user's money will be returned.

5. Check that when enough money was deposited, the beverage will be dispensed, and any extra change will be returned.



JUnit

Finished after 0.005 seconds

Runs: 3/3 Errors: 0 Failures: 0

- edu.ncsu.csc326.coffeemaker.CoffeeMakerTest [Ru

Failure Trace

```
    public void testMakeCoffee() {
        cm.addRecipe(r1);
        assertEquals(25, cm.makeCoffee(0, 75));

        cm.addRecipe(r2); //r2 costs 75
        assertEquals(20, cm.makeCoffee(0, 20));

        System.out.println("Before purchasing: \n"+cm.checkInventory());

        assertEquals(100, cm.makeCoffee(1, 100));

        try {
            cm.addInventory("0","0","0","5");
        } catch (InventoryException e) {
            e.printStackTrace();
        }
        // beverage is made, change is returned
        assertEquals(25, cm.makeCoffee(1, 100));
        System.out.println("After purchasing: \n"+cm.checkInventory());
    }
```

Failed to make a coffee due to lack of chocolate, full amount of money is returned.
r2 requires 20 units of chocolate, inventory has 15.

5 units of chocolate is added to the inventory. There is enough ingredient to make r2 drink.

Drink is made. R2 costs 75. 100 units of money is taken. 25 units of change is returned.

Problems Javadoc Declaration Console Error Log

```
<terminated> CoffeeMakerTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.7.0_75.jdk/Contents/Home/bin/java (Oct 16, 2017, 11:21:04 PM)
Before purchasing:
Coffee: 18
Milk: 14
Sugar: 14
Chocolate: 15

After purchasing:
Coffee: 21
Milk: 13
Sugar: 13
Chocolate: 0
```

Summary:

Bugs discovered: 5

Test cases created:

- William Rosenberger: 13
- Christina Yu: 12