Inf2-SEPP 2024-25
Tutorial 6 (Week 10)
Testing and Test Coverage

Study this tutorial sheet and make notes of your answers BEFORE the tutorial.

1 Introduction

The purpose of this tutorial is to get you to put into practice concepts that you have studied in the lectures on testing: unit testing, system testing, test coverage. It also gives you a glimpse into some of the important types of tasks that will be involved in your Coursework 3.

To be able to tackle this tutorial, apart from the lecture material you will also need to be familiar with the JUnit5 automatic testing framework for Java. Here is a handy tutorial: `https://www.vogella.com/tutorials/JUnit/article.html`.

2 Task 1: Unit Testing and Test Coverage

In the last tutorial, you started work on a coffee calculator tool. You have now been assigned to write tests for this tool. Within it, you have the following method:

```
int getReducedCalories(int initialCalories, boolean skimmedMilk, boolean sugarFreeSyrup){
     if ((initialCalories<9) && (initialCalories>250))
          return -1;
     if (skimmedMilk==true)
          return initialCalories - 70;
     if (sugarFreeSyrup==true)
          return initialCalories - 80;
     return initialCalories;
}
```

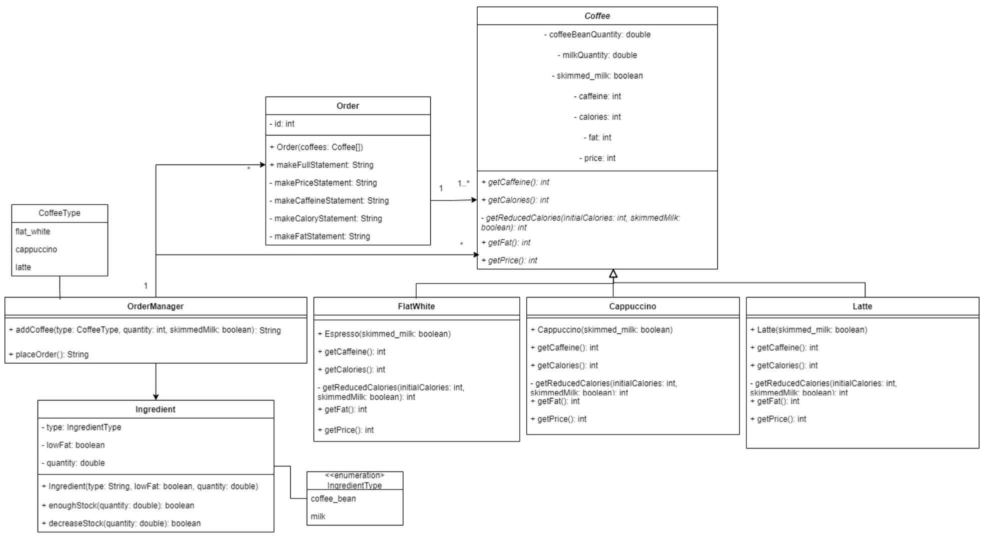
You also have access to a requirements specification document, which includes a requirement about allowing coffee shop employees to calculate the reduction in calories (natural number) of an unsweetened cup of coffee when one or more of its contents are replaced with reduced-calory alternatives, based on the initial number of calories of the given coffee type. The number of calories of a cup of coffee (considering normal content quantities), depending on its type, can only vary between 9 and 250 no matter the contents (type of milk, type of syrup), and the employee needs to be notified in case he/she enters initial calory values that would ouput a result other than in this interval. Moreover, the employee should be notified when they have not selected at least one replacement content. Skimmed milk reduces the calories by 70 and sugar free syrup reduces them by 80.

1. Considering only the specification and not looking at the structure of the code (i.e. black box testing), write the assert statements from within JUnit5 unit tests for this method. Make sure that you test for any incorrect inputs as well as for all (or at least as many as you can think of!) possible combinations of classes of values. Also, have the tests return intuitive messages when failing. You can make the assumption that incorrect inputs should lead to an output of -1.
2. (optional) Set up a project on your IDE, create a Calculator class, and copy the above method in it. Then, set up a test class and write your tests for this class within it. Each test should only be testing one set of inputs, i.e. have one assert statement. Run the tests.
3. What are the statement and branch coverage (considering only the code of the method) of your tests? Calculate it on paper.
4. Is it possible to ever reach 100% statement and 100% branch coverage for this code? Write any additional tests (this time white-box testing, as we are looking at the code) to maximise the statement and branch coverage that you can get, then run them.
5. Which of the tests in your final set fail? Why? How would you fix the code? If you have implemented the code (step 2), go ahead and make this change. How can you decide whether the code is now correct?

3 Task 2: System Testing

You are now much more advanced in your work on the coffee calculator tool, and this tool has been included in a coffee shop system. The coffee shop has made a decision to no longer use syrups due to their negative impact on health. So far, it is only selling three types of coffee: flat white, cappuccino and latte.

You are provided with a class diagram for the system, provided below. It omits the typically looking getter and setter methods. Moreover, you are given the following description for the main success scenarios and alternative scenarios of the "Place Order" use case:


Coffee
- coffeeBeanQuantity: double
- milkQuantity: double
- skimmed_milk: boolean
- caffeine: int
- calories: int
- fat: int
- price: int
+ getCaffeine(): int
+ getCalories(): int
- getReducedCalories(initialCalories: int, skimmedMilk: boolean): int
+ getFat(): int
+ getPrice(): int
Order
- id: int
+ Order(coffees: Coffee[])
+ makeFullStatement: String
- makePriceStatement: String
- makeCaffeineStatement: String
- makeCaloryStatement: String
- makeFatStatement: String
*
1
1..*
*
1
CoffeeType
flat_white
cappuccino
latte
OrderManager
+ addCoffee(type: CoffeeType, quantity: int, skimmedMilk: boolean): String
+ placeOrder(): String
FlatWhite
+ Espresso(skimmed_milk: boolean)
+ getCaffeine(): int
+ getCalories(): int
- getReducedCalories(initialCalories: int, skimmedMilk: boolean): int
+ getFat(): int
+ getPrice(): int
Cappuccino
+ Cappuccino(skimmed_milk: boolean)
+ getCaffeine(): int
+ getCalories(): int
- getReducedCalories(initialCalories: int, skimmedMilk: boolean): int
+ getFat(): int
+ getPrice(): int
Latte
+ Latte(skimmed_milk: boolean)
+ getCaffeine(): int
+ getCalories(): int
- getReducedCalories(initialCalories: int, skimmedMilk: boolean): int
+ getFat(): int
+ getPrice(): int
Ingredient
- type: IngredientType
- lowFat: boolean
- quantity: double
+ Ingredient(type: String, lowFat: boolean, quantity: double)
+ enoughStock(quantity: double): boolean
+ decreaseStock(quantity: double): boolean
<<enumeration>>
IngredientType
coffee_bean
milk

```
Main Success Scenario:
     1. The customer requests a certain quantity of a type of coffee (flat white,
     cappuccino or latte), and indicates if they would like skimmed milk or not.
     2. The system confirms the coffee type with required quantity was requested
     3. The customer chooses to place the order.
     4. The system confirms the order with a full statement including the order id,
     caffeine, calories (considering the chosen type of milk), fat, as well as price.
Extensions:
     2a. If the quantity is an invalid number or the coffee type provided is not
      supported by the system, the system returns an error message "Invalid inputs
      provided". The use case terminates.
     3a. If the customer would like to add more coffees, return to MSS step 1
     4a. If there is not enough stock of ingredients (coffee beans and/or milk
      of the wanted type) left for making that quantity of the wanted type of coffee,
      the system returns an error message "Not enough stock left to make
      one or more coffees". The use case terminates.
     4b. If the order contains several coffees, and there is not enough stock
      of ingredients (coffee beans and/or milk
      of the wanted type) left for making the quantity of any of the wanted types
      of coffee,  the system returns an error message "Not enough stock left to
      make one or more coffees". The use case terminates.
      4c If the order contains several coffees, and there is enough stock for all, the
       system includes in the statement the order id, the total levels for caffeine,
      calories, fat, and the total price.
```

1. Following the provided class diagram, figure out how the implementation would support the above use case with all of its scenarios.
2. (Optional) Write the code to match the class diagram and address the use case with all of its scenarios.
3. Write system tests to test each possible scenario in the use case. As the number of coffee types ordered can be infinite, you are allowed to use a limit of 2 coffee types for testing purposes. Moreover, you are encouraged to test with different coffee types, quantities and milk types, to try to achieve as much coverage (statement and branch) as possible of the code.

Cristina Adriana Alexandru. 20 March 2025.