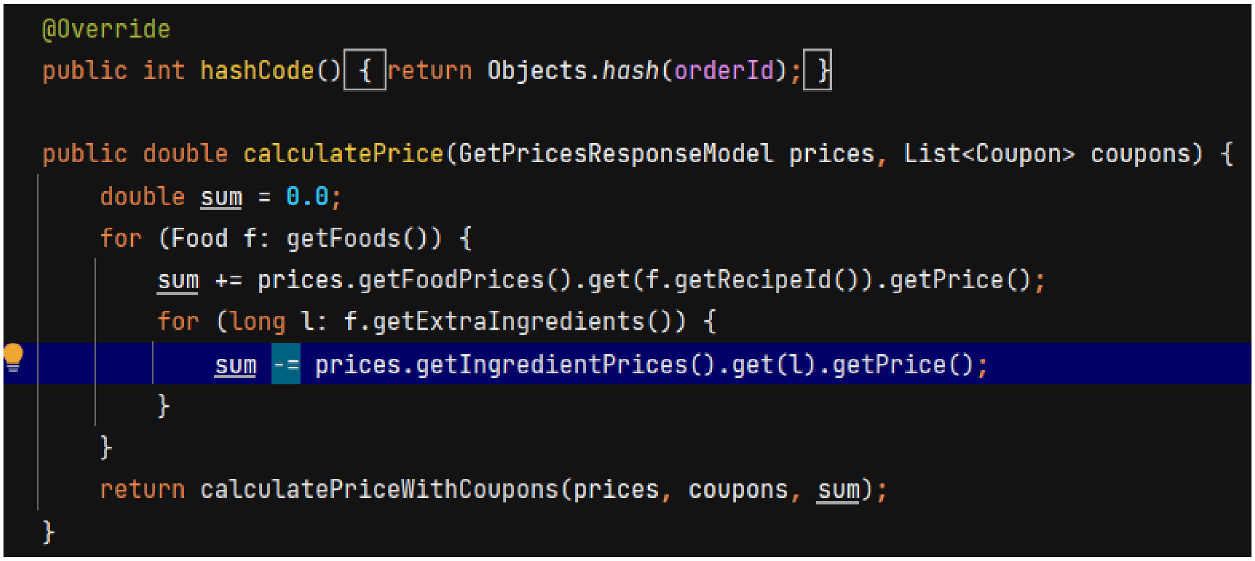
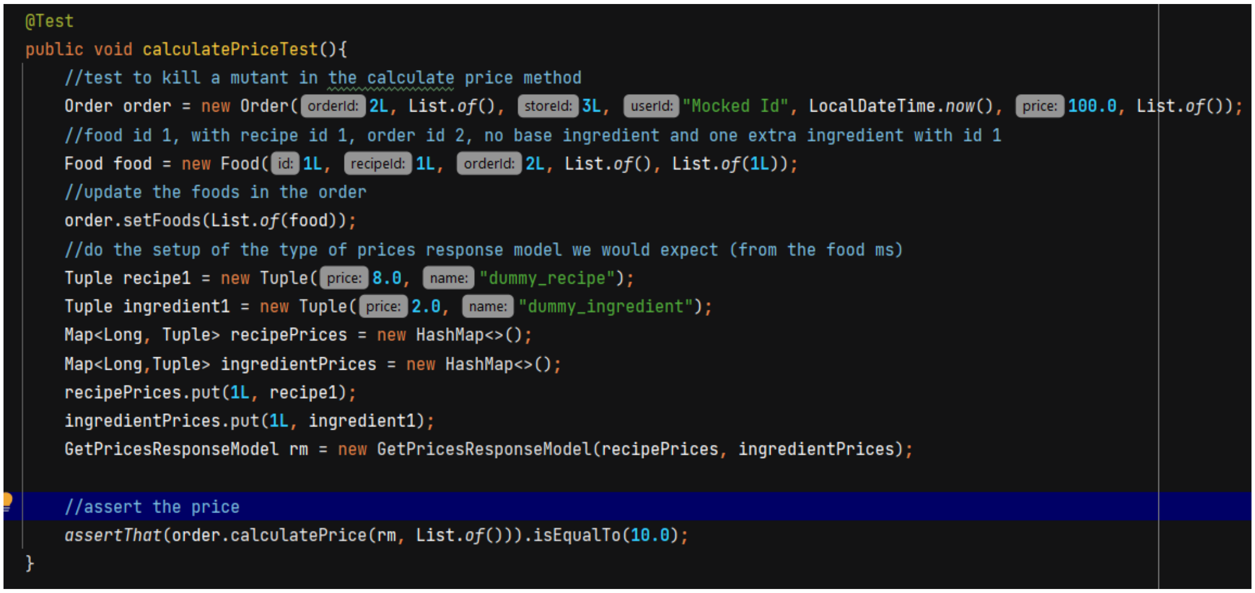
Manual analysis and correction of 4 mutants in core classes

1) First we chose to take a closer look at the `Order` class, since it is probably one of the most central parts of our application. It is responsible for the core logic of order processing. In our search, we found an uncaught mutant in the `calculatePrice` method, by changing a mathematical operator in the extra ingredient price calculation.



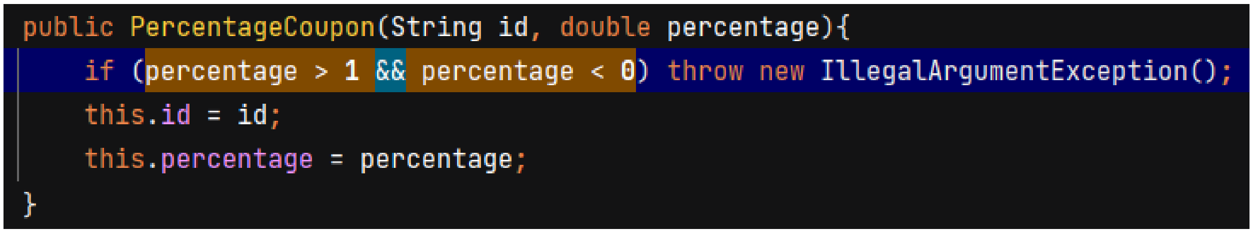
```
@Override
public int hashCode() { return Objects.hash(orderId); }

public double calculatePrice(GetPricesResponseModel prices, List<Coupon> coupons) {
    double sum = 0.0;
    for (Food f: getFoods()) {
        sum += prices.getFoodPrices().get(f.getRecipeId()).getPrice();
        for (long l: f.getExtraIngredients()) {
            sum -= prices.getIngredientPrices().get(l).getPrice();
        }
    }
    return calculatePriceWithCoupons(prices, coupons, sum);
}
```

In response, we wrote a new unit test, `calculatePriceTest`, that makes sure that the price is always calculated correctly, regardless of what extra ingredients a food has.



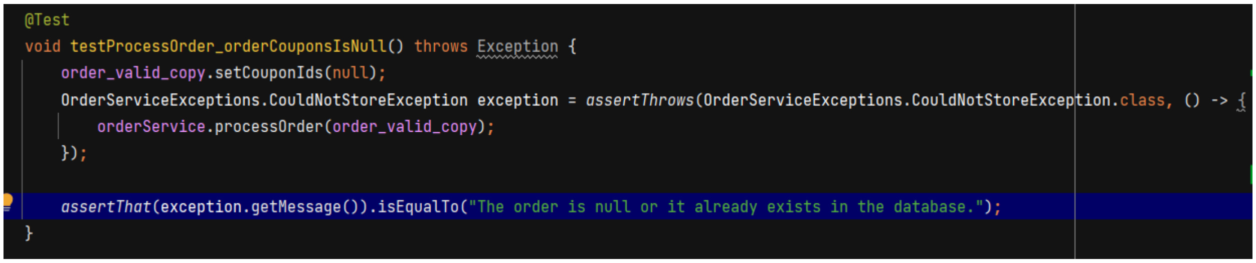
```
@Test
public void calculatePriceTest(){
    //test to kill a mutant in the calculate price method
    Order order = new Order( orderId: 2L, List.of(),  storeId: 3L,  userId: "Mocked Id", LocalDateTime.now(),  price: 100.0, List.of());
    //food id 1, with recipe id 1, order id 2, no base ingredient and one extra ingredient with id 1
    Food food = new Food( id: 1L,  recipeId: 1L,  orderId: 2L, List.of(), List.of(1L));
    //update the foods in the order
    order.setFoods(List.of(food));
    //do the setup of the type of prices response model we would expect (from the food ms)
    Tuple recipe1 = new Tuple( price: 8.0,  name: "dummy_recipe");
    Tuple ingredient1 = new Tuple( price: 2.0,  name: "dummy_ingredient");
    Map<Long, Tuple> recipePrices = new HashMap<>();
    Map<Long,Tuple> ingredientPrices = new HashMap<>();
    recipePrices.put(1L, recipe1);
    ingredientPrices.put(1L, ingredient1);
    GetPricesResponseModel rm = new GetPricesResponseModel(recipePrices, ingredientPrices);

    //assert the price
    assertThat(order.calculatePrice(rm, List.of())).isEqualTo(10.0);
}
```

2) The second class we looked into was PercentageCoupon, since it is also involved in price calculations and it is thus a critical part of the system. The mutant we found was in the input validation of the constructor itself.



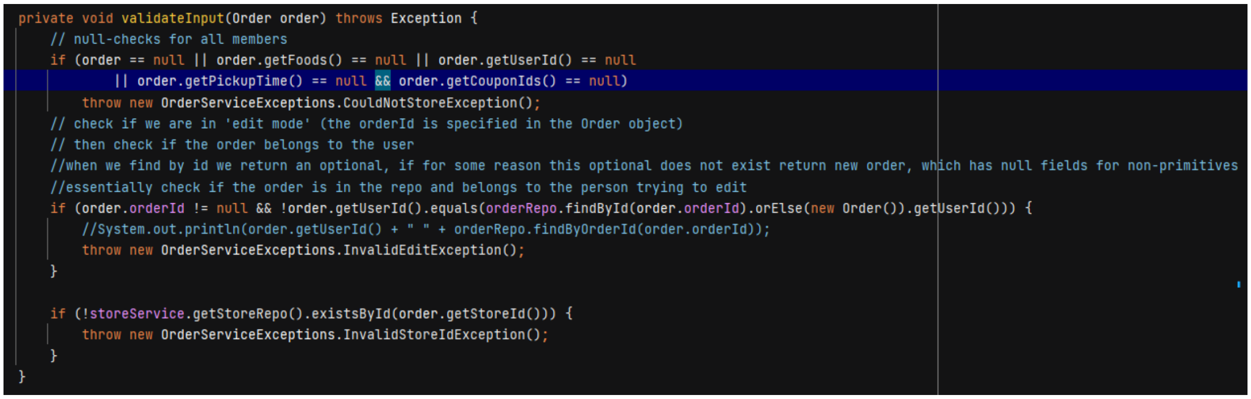
```java
public PercentageCoupon(String id, double percentage){
    if (percentage > 1 && percentage < 0) throw new IllegalArgumentException();
    this.id = id;
    this.percentage = percentage;
}
```

To counter that, we wrote a new test, `testThrowsIllegal`, which checks that the coupon always has valid attributes.



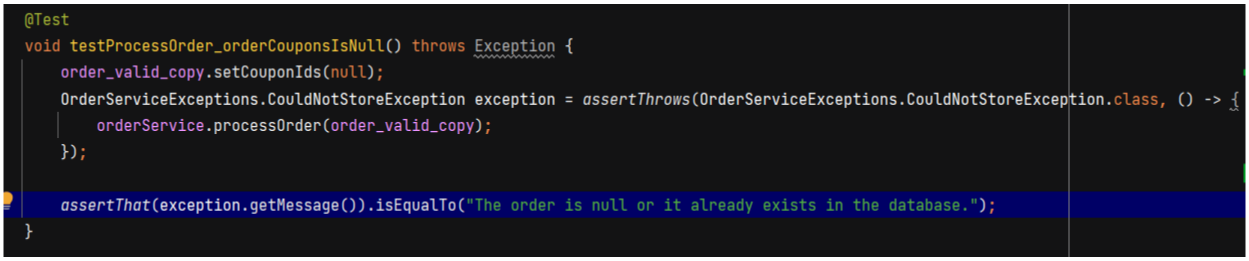
```java
@Test
void testProcessOrder_orderCouponsIsNull() throws Exception {
    order_valid_copy.setCouponIds(null);
    OrderServiceExceptions.CouldNotStoreException exception = assertThrows(OrderServiceExceptions.CouldNotStoreException.class, () -> {
        orderService.processOrder(order_valid_copy);
    });

    assertThat(exception.getMessage()).isEqualTo("The order is null or it already exists in the database.");
}
```

3) Next up, we turned our attention to the `OrderService` class, more specifically to the `processOrder` method. I think the name is pretty suggestive when it comes to showing the importance of this one. The mutant laid in the input validation, more specifically in the null-checks of the order object.



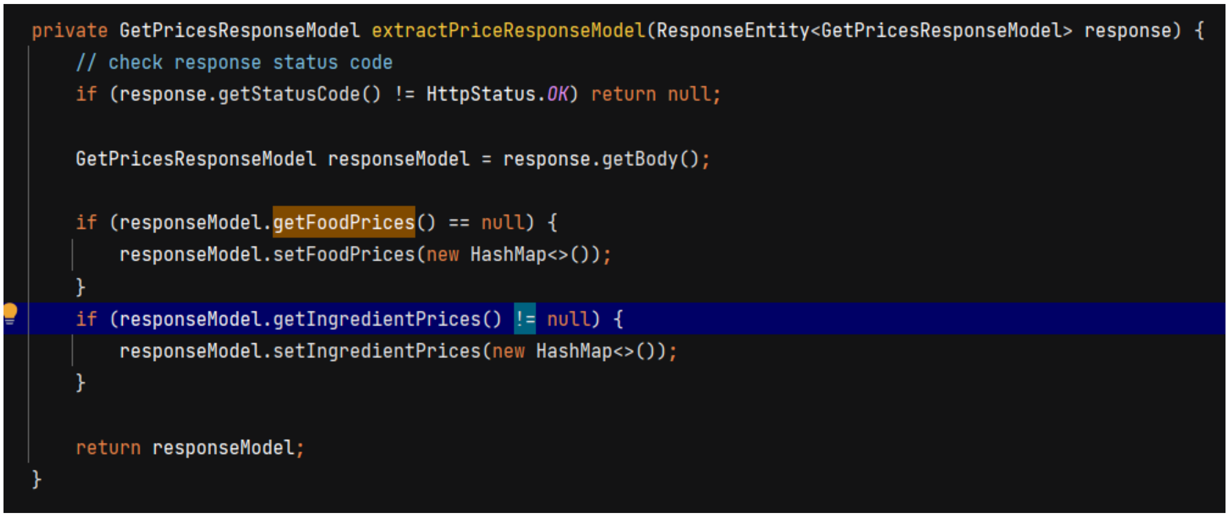
```java
private void validateInput(Order order) throws Exception {
    // null-checks for all members
    if (order == null || order.getFoods() == null || order.getUserId() == null
            || order.getPickupTime() == null && order.getCouponIds() == null)
        throw new OrderServiceExceptions.CouldNotStoreException();
    // check if we are in 'edit mode' (the orderId is specified in the Order object)
    // then check if the order belongs to the user
    //when we find by id we return an optional, if for some reason this optional does not exist return new order, which has null fields for non-primitives
    //essentially check if the order is in the repo and belongs to the person trying to edit
    if (order.orderId != null && !order.getUserId().equals(orderRepo.findById(order.orderId).orElse(new Order()).getUserId())) {
        //System.out.println(order.getUserId() + " " + orderRepo.findByOrderId(order.orderId));
        throw new OrderServiceExceptions.InvalidEditException();
    }

    if (!storeService.getStoreRepo().existsById(order.getStoreId())) {
        throw new OrderServiceExceptions.InvalidStoreIdException();
    }
}
```

We corrected this with a new test, `testProcessOrder_orderCouponIsNull`.



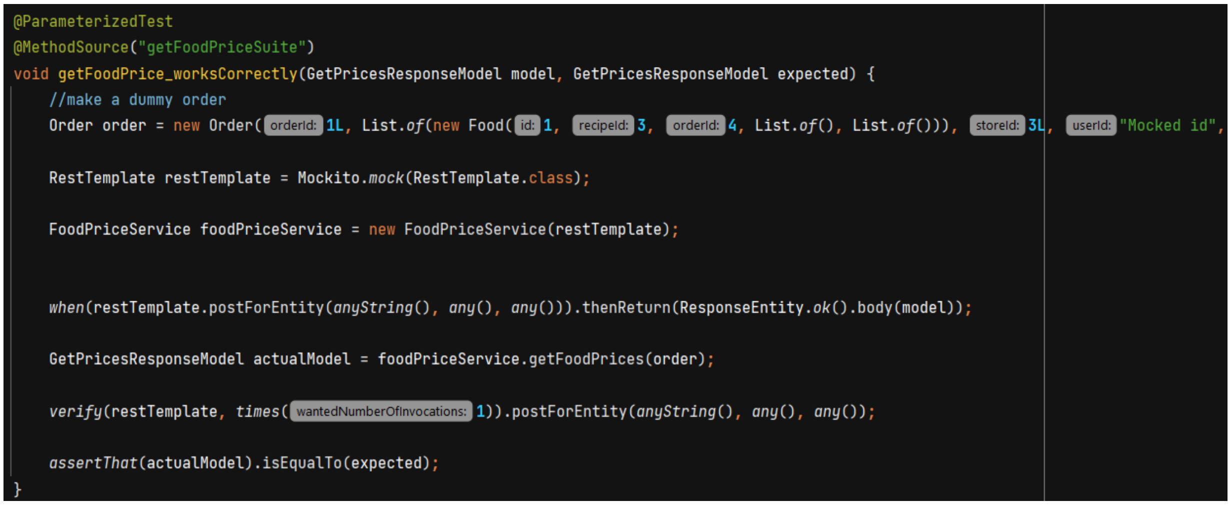
```java
@Test
void testProcessOrder_orderCouponsIsNull() throws Exception {
    order_valid_copy.setCouponIds(null);
    OrderServiceExceptions.CouldNotStoreException exception = assertThrows(OrderServiceExceptions.CouldNotStoreException.class, () -> {
        orderService.processOrder(order_valid_copy);
    });

    assertThat(exception.getMessage()).isEqualTo("The order is null or it already exists in the database.");
}
```

4) Last but not least, we analyzed the `extractPriceResponseModel` method in the `FoodPriceService` class. We found an uncaught mutant in the form of a logical operator that had the potential to break our input validation.



```
private GetPricesResponseModel extractPriceResponseModel(ResponseEntity<GetPricesResponseModel> response) {
    // check response status code
    if (response.getStatusCode() != HttpStatus.OK) return null;

    GetPricesResponseModel responseModel = response.getBody();

    if (responseModel.getFoodPrices() == null) {
        responseModel.setFoodPrices(new HashMap<>());
    }
    if (responseModel.getIngredientPrices() != null) {
        responseModel.setIngredientPrices(new HashMap<>());
    }

    return responseModel;
}
```

We wrote a test called `getFoodPrice_worksCorrectly` and edited its referenced setup function `getFoodPriceSuite` to kill this mutant.



```
@ParameterizedTest
@MethodSource("getFoodPriceSuite")
void getFoodPrice_worksCorrectly(GetPricesResponseModel model, GetPricesResponseModel expected) {
    //make a dummy order
    Order order = new Order( orderId: 1L, List.of(new Food( id: 1,  recipeId: 3,  orderId: 4, List.of(), List.of())),  storeId: 3L,  userId: "Mocked id",

    RestTemplate restTemplate = Mockito.mock(RestTemplate.class);

    FoodPriceService foodPriceService = new FoodPriceService(restTemplate);


    when(restTemplate.postForEntity(anyString(), any(), any())).thenReturn(ResponseEntity.ok().body(model));

    GetPricesResponseModel actualModel = foodPriceService.getFoodPrices(order);

    verify(restTemplate, times( wantedNumberOfInvocations: 1)).postForEntity(anyString(), any(), any());

    assertThat(actualModel).isEqualTo(expected);
}
```