# Code Review

The main area of code I struggled with was the order validation method. It was filled with repetition and redundancies which makes it go against some of the essential software protocols of DRY (Do Not Repeat). Prior to reaching this learning outcome, in LO3 and LO4 I already found and fixed the following areas in the code:

- Delivery fee: The delivery fee was not being properly added to the total price of pizzas resulting in every valid order being classified as invalid.
- Expiry date: The credit card expiry date was being considered as being at the beginning rather than the end of the month resulting in orders made in the same month as the expiry date being marked as invalid.

This was the code for my order validation method at the start of approaching this learning outcome (I have not included the full method in screenshots as it was long and repetitive):

```
@PostMapping("/validateOrder")  Alice de Jongh +1 *
public ResponseEntity<OrderValidationResult> validateOrder(@RequestBody Order order) {

    OrderValidationResult result = new OrderValidationResult();

    try {
        if (order == null
                || order.getCreditCardInformation() == null
                || order.getPizzasInOrder() == null
                || order.getOrderDate() == null) {
            return ResponseEntity.badRequest().build();
        }

        if (order.getCreditCardInformation().getCreditCardNumber() == null
                || order.getCreditCardInformation().getCreditCardExpiry() == null
                || order.getCreditCardInformation().getCvv() == null) {
            return ResponseEntity.badRequest().build();
        }

        if (order.getPizzasInOrder().isEmpty()) {
            result.setOrderStatus(OrderStatus.INVALID);
            result.setOrderValidationCode(OrderValidationCode.EMPTY_ORDER);
            return ResponseEntity.ok(result);
        }

        try {
            String expiry = order.getCreditCardInformation().getCreditCardExpiry();

            if (!expiry.matches( regex: "\\d{2}/\\d{2}")) {
                throw new IllegalArgumentException();
            }

            YearMonth expiryMonth = YearMonth.parse(expiry, DateTimeFormatter.ofPattern("MM/yy"));
            YearMonth currentMonth = YearMonth.now();

            if (expiryMonth.isBefore(currentMonth)) {
                result.setOrderStatus(OrderStatus.INVALID);
                result.setOrderValidationCode(OrderValidationCode.EXPIRY_DATE_INVALID);
                return ResponseEntity.ok(result);
            }

        } catch (Exception e) {
            result.setOrderStatus(OrderStatus.INVALID);
            result.setOrderValidationCode(OrderValidationCode.EXPIRY_DATE_INVALID);
            return ResponseEntity.ok(result);
        }

        if (!order.getCreditCardInformation().getCreditCardNumber().matches( regex: "\\d{16}")) {
            result.setOrderStatus(OrderStatus.INVALID);
            result.setOrderValidationCode(OrderValidationCode.CARD_NUMBER_INVALID);
            return ResponseEntity.ok(result);
        }

        if (!order.getCreditCardInformation().getCvv().matches( regex: "\\d{3}")) {
            result.setOrderStatus(OrderStatus.INVALID);
            result.setOrderValidationCode(OrderValidationCode.CVV_INVALID);
            return ResponseEntity.ok(result);
        }
```

```
        int calculatedTotal = order.getPizzasInOrder().stream() Stream<Pizza>
                .mapToInt(Order.Pizza::getPriceInPence) IntStream
                .sum();

        int expectedTotal = calculatedTotal + SystemConstants.ORDER_CHARGE_IN_PENCE;

        if (!Integer.valueOf(expectedTotal).equals(order.getPriceTotalInPence())) {
            result.setOrderStatus(OrderStatus.INVALID);
            result.setOrderValidationCode(OrderValidationCode.TOTAL_INCORRECT);
            return ResponseEntity.ok(result);
        }

        result.setOrderStatus(OrderStatus.VALID);
        result.setOrderValidationCode(OrderValidationCode.NO_ERROR);
        return ResponseEntity.ok(result);

    } catch (Exception e) {
        e.printStackTrace();
        result.setOrderStatus(OrderStatus.INVALID);
        result.setOrderValidationCode(OrderValidationCode.GENERIC_ERROR);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(result);
    }
}
```

I chose a checklist-based inspection (Checklist Based Testing, n.d.) approach to this code review as a sole developer with limited resources (time specifically) and it is particularly suitable for order validation as the requirements can be approached as a checklist due to each requirement being able to be tested independently of the others.

## Review Criteria

| Criteria | Level prior to code review |
|---|---|
| Readability of the logic | Moderate – easy to follow but repetitive |
| Testability of the logic | High – each requirement can be tested independently like the ILP auto-marker |
| Handling of null and malformed inputs | This was identified previously and null checks were added to each if statement corresponding to a requirement |
| Handling boundary conditions | This is accessed and errors fixed in prior learning outcomes |

Following the steps outlined in the Test Automation lecture for Review Activities:
'Setup review, prepare code and distribute code/tests' is unnecessary as the sole developer on this project I have full access to the code and will be reviewing it myself.

The 'Check code' stage was accomplished in LO3 where I tested every requirement in my validateOrder() method with integration tests and I wrote up the code review and the fixes I implemented in the **Testing Document** as part of the 'write review' stage.

Here I will further add to the 'Write Review' stage as I will not just be accessing the correctness of my code in alignment with the ILP specification; I will be accessing the readability and quality of the code. As established earlier, there is high repetition within the validateOrder() method mainly due to the repeated patterns for setting status and validation codes. The method is also long and dense which needlessly increases complexity during review.

There was no discussion stage as I did not involve an outside reviewer. For the ‘Make to-do list’ stage, I used the Notes app on my phone that I could access on my laptop as well. This made it so that I could note down any ideas throughout the day when I was away from my laptop and then be able to access it easily when it came time to implement those changes.

## Make code changes

Below is the improved code.

```
@PostMapping("/validateOrder")  Alice de Jongh +1 *
public ResponseEntity<OrderValidationResult> validateOrder(@RequestBody Order order) {
    OrderValidationResult result = new OrderValidationResult();
    try {
        if (order == null || order.getCreditCardInformation() == null
                || order.getPizzasInOrder() == null || order.getOrderDate() == null)
            return ResponseEntity.badRequest().build();
        var card = order.getCreditCardInformation();

        if (card.getCreditCardNumber() == null
                || card.getCreditCardExpiry() == null
                || card.getCvv() == null)
            return ResponseEntity.badRequest().build();

        if (order.getPizzasInOrder().isEmpty())
            return invalid(result, OrderValidationCode.EMPTY_ORDER);

        if (!isValidExpiry(card.getCreditCardExpiry()))
            return invalid(result, OrderValidationCode.EXPIRY_DATE_INVALID);

        if (!card.getCreditCardNumber().matches( regex: "\\d{16}"))
            return invalid(result, OrderValidationCode.CARD_NUMBER_INVALID);

        if (!card.getCvv().matches( regex: "\\d{3}"))
            return invalid(result, OrderValidationCode.CVV_INVALID);

        if (order.getPizzasInOrder().size() > SystemConstants.MAX_PIZZAS_PER_ORDER)
            return invalid(result, OrderValidationCode.MAX_PIZZA_COUNT_EXCEEDED);
```

```
        LocalDate orderDate;
        try {
            orderDate = LocalDate.parse(order.getOrderDate());
        } catch (DateTimeParseException e) {
            return ResponseEntity.badRequest().build();
        }

        DayOfWeek day = orderDate.getDayOfWeek();
        String restaurantName = null;

        for (Order.Pizza pizza : order.getPizzasInOrder()) {
            Restaurant r = findRestaurantByPizza(pizza.getName());

            if (r == null)
                return invalid(result, OrderValidationCode.PIZZA_NOT_DEFINED);

            if (!isPriceValid(pizza, r))
                return invalid(result, OrderValidationCode.PRICE_FOR_PIZZA_INVALID);

            if (restaurantName == null)
                restaurantName = r.getName();
            else if (!restaurantName.equals(r.getName()))
                return invalid(result, OrderValidationCode.PIZZA_FROM_MULTIPLE_RESTAURANTS);

            if (!r.getOpeningDays().contains(day.toString()))
                return invalid(result, OrderValidationCode.RESTAURANT_CLOSED);
        }

        int expectedTotal = order.getPizzasInOrder().stream() Stream<Pizza>
```

```
        if (expectedTotal != order.getPriceTotalInPence())
            return invalid(result, OrderValidationCode.TOTAL_INCORRECT);

        result.setOrderStatus(OrderStatus.VALID);
        result.setOrderValidationCode(OrderValidationCode.NO_ERROR);
        return ResponseEntity.ok(result);

    } catch (Exception e) {
        result.setOrderStatus(OrderStatus.INVALID);
        result.setOrderValidationCode(OrderValidationCode.GENERIC_ERROR);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(result);
    }
}

private ResponseEntity<OrderValidationResult> invalid(  10 usages  new *
        OrderValidationResult result, OrderValidationCode code) {
    result.setOrderStatus(OrderStatus.INVALID);
    result.setOrderValidationCode(code);
    return ResponseEntity.ok(result);
}

private boolean isValidExpiry(String expiry) {  1 usage  new *
    if (!expiry.matches( regex: "\\d{2}/\\d{2}")) return false;
    YearMonth exp = YearMonth.parse(expiry, DateTimeFormatter.ofPattern("MM/yy"));
    return !exp.isBefore(YearMonth.now());
}
```

Here the full method plus two new helper methods can be fully seen whereas before I had to cut pieces out to make it fit into three pictures.

After reviewing the validateOrder method, I refactored it to improve readability and reduce unnecessary repetition while keeping all logic inside the controller and preserving the behaviour required by the ILP specification. The original method worked functionally but was long and repetitive due to the same response logic being repeated across many validation branches. I reorganised the method so that related checks are grouped together (request shape validation, credit card validation, pizza constraints, restaurant checks, and total price validation), which makes the flow of the logic much clearer and easier to map back to individual requirements.

The early-return structure was deliberately kept so that each invalid order triggers exactly one validation error, matching the behaviour of the ILP auto-marker and supporting isolated testing of each rule. Defensive checks for malformed requests remain at the start of the method to clearly separate HTTP-level errors from order validation errors. Overall, the refactored method is shorter, easier to read, and easier to test, while remaining fully adhering to the coursework requirements.

## Bibliography

*Checklist Based Testing*. (n.d.). Retrieved from QATestLab: https://qatestlab.com/resources/knowledge-center/checklist-based/