Assignment 2

Introduction and overall choices:

The tool we have decided to use for analyzing our current version of the application is Metrics Tree[1]. This is an IntelliJ plugin that allows to view all sorts of code smells and parameters used to measure them. When computing the code metrics, we found that most of our issues were related to the following: long methods, too many fields and coupling.

We chose to ignore the coupling "issues" because they were related to the coupling between controllers and services. A layered architecture like the one we have causes the controllers and services to be very closely coupled simply because they are components in the chain of processing what comes into an endpoint. The controller manages the endpoints and responding if needed, and the service does the validation and processing, by having controllers relegate the data they collected. Inside the service we may also call repositories to store some data. If we aimed to reduce the coupling, we would break the architecture. It makes sense that these classes are coupled, therefore we will not make changes to them.

So, we will mostly work with long methods (LOC, occurs when validating an order, calculating prices, and other more complex functionalities), on complex method (cyclomatic complexity, also when doing validation) and with too many fields in the classes.

Long methods:

First smell ProcessOrder*(...):* This method in the order service was before the refactoring the longest method in the project, 75 lines of code. This is a logical choice for refactoring due to maintainability and clean code, as well as functionality. Currently, the method does two things: validation and saving the order. However, validation is on many aspects: user id, validity of store, price calculation, etc. We choose to split this functionality into smaller methods to have a clear differentiation of responsibilities. Also, if we later change how we validate a store, or some other criteria changes, we will be able to find it easily and cause less trouble when changing. Below a screenshot of the code metrics before the refactoring:

[1] https://plugins.jetbrains.com/plugin/13959-metricstree

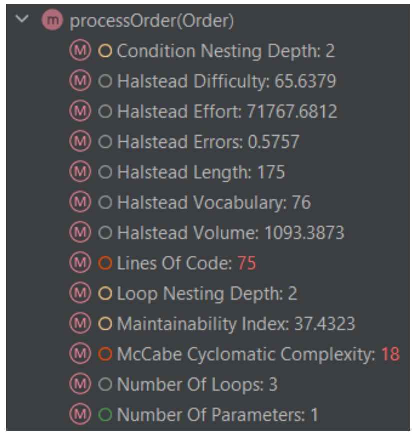


Fig 1: Code metrics before refactoring *ProcessOrder( )*: notice especially LOC and CC

We note that 75 lines of code is far above the acceptable threshold of 30-40 lines. Additionally, we notice that the cyclomatic complexity is at 18, which is also excessively high considering acceptable levels of around 5-10. Therefore, we choose to extract methods.

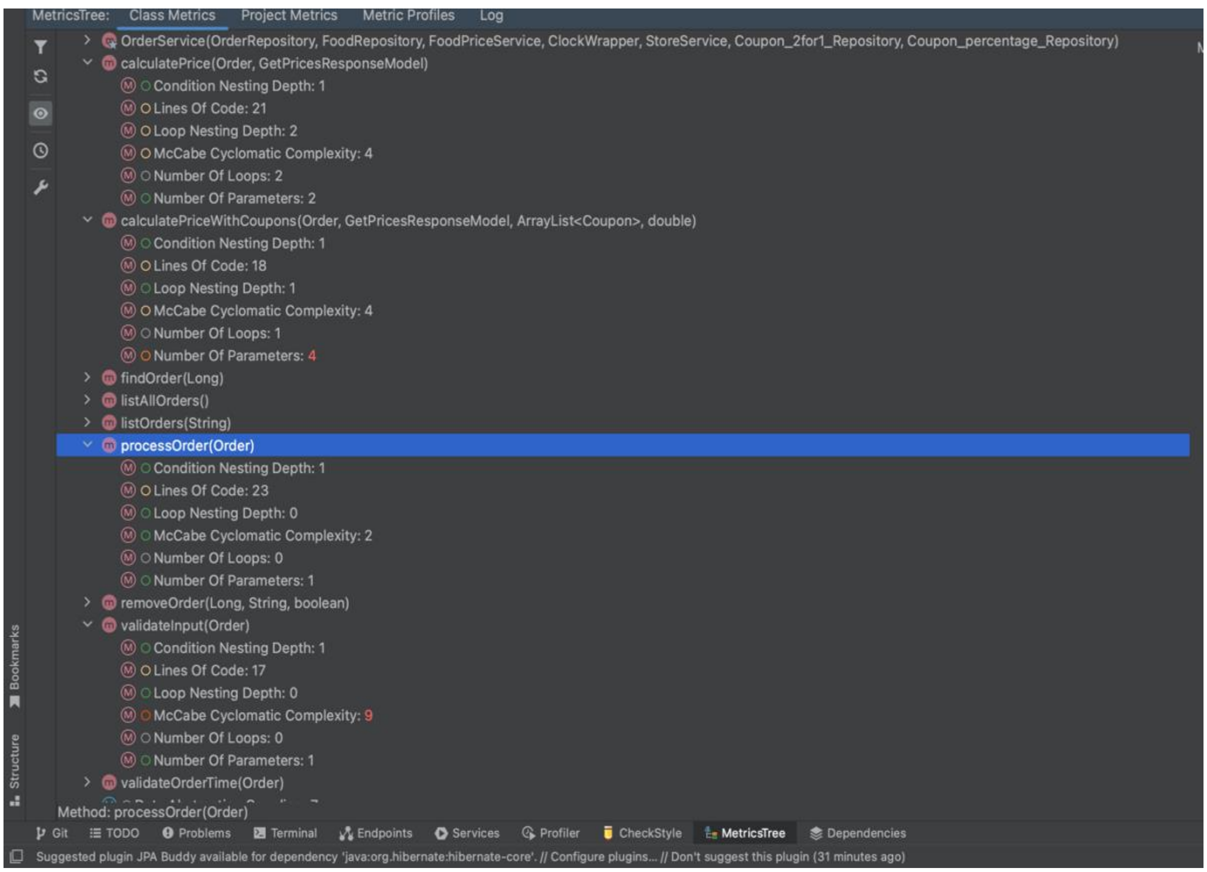


Fig 2: Code metrics after refactoring *ProcessOrder( )*: we split the method

```java
public Order processOrder(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 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 field
    //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 InvalidEditException();
    }

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

    //check if the selected pickup time is 30 minutes or more in the future
    LocalDateTime current = clockWrapper.getNow();

    if (order.getPickupTime().isBefore(current.plusMinutes(30)))
        throw new TimeInvalidException();

    GetPricesResponseModel prices = foodPriceService.getFoodPrices(order); // get prices
    if (prices == null)
        //some food does not exist or something else went wrong in the food ms communication
        throw new FoodInvalidException();
```

```java
    ArrayList<Coupon> coupons = new ArrayList<>(coupon_percentage_repository.findAllById(order.couponIds));
    coupons.addAll(coupon_2for1_repository.findAllById(order.couponIds));
    // this list only contains validated coupons, no need for additional checks
    order.couponIds.clear(); // clear the list, so we can send only the used one back

    //get the base price of the order
    double sum = 0.0;
    for (Food f: order.getFoods()) {
        sum += prices.getFoodPrices().get(f.getRecipeId()).getPrice();
        for (long l: f.getExtraIngredients()) {
            sum += prices.getIngredientPrices().get(l).getPrice();
        }
    }

    if (coupons.isEmpty()) { // If coupon list is empty, just add all ingredients and recipes
        final double EPS = 1e-6;
        if (Math.abs(order.price - sum) > EPS) {
            throw new PriceNotRightException("Price is not right");
        }

        return orderRepo.save(order);
    }

    double minPrice = Double.MAX_VALUE;
    order.couponIds.add("0");

    for (Coupon c: coupons) {
```

```java
    for (Coupon c: coupons) {
        //iterate over the list of valid coupons
        double price = c.calculatePrice(order, prices, sum);

        if (Double.compare(price, minPrice) < 0) {
            minPrice = price;
            //set the first element in the coupon ids to the coupon used
            //order.couponIds.clear();
            order.couponIds.set(0, c.getId());
        }
    }

    final double EPS = 1e-6;
    if (Math.abs(order.price - minPrice) > EPS) {
        throw new PriceNotRightException("Price is not right");
    }

    return orderRepo.save(order);
}
```

Fig3, 4, 5: Code before refactoring

```java
public Order processOrder(Order order) throws Exception {
    validateInput(order);

    validateOrderTime(order);

    GetPricesResponseModel prices = foodPriceService.getFoodPrices(order); // get prices
    if (prices == null)
        //some food does not exist or something else went wrong in the food ms communication
        throw new OrderServiceExceptions.FoodInvalidException();

    return calculatePrice(order, prices);
}

/PMD/
private Order calculatePrice(Order order, GetPricesResponseModel prices) throws OrderServiceExceptions.PriceNotRightException {
    ArrayList<Coupon> coupons = new ArrayList<>(coupon_percentage_repository.findAllById(order.couponIds));
    coupons.addAll(coupon_2for1_repository.findAllById(order.couponIds));
    // this list only contains validated coupons, no need for additional checks
    order.couponIds.clear(); // clear the list, so we can send only the used one back
    //get the base price of the order
    double sum = order.calculatePrice(prices, coupons);

    //sum = calculatePriceWithCoupons(order, prices, coupons);
    final double EPS = 1e-6;
    if (Math.abs(order.price - sum) > EPS) {
        throw new OrderServiceExceptions.PriceNotRightException("Price is not right");
    }
    return orderRepo.save(order);
}
```

```java
private void validateOrderTime(Order order) throws OrderServiceExceptions.TimeInvalidException {
    //check if the selected pickup time is 30 minutes or more in the future
    LocalDateTime current = clockWrapper.getNow();

    if (order.getPickupTime().isBefore(current.plusMinutes(30)))
        throw new OrderServiceExceptions.TimeInvalidException();
}

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
    //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.existsById(order.getStoreId())) {
        throw new OrderServiceExceptions.InvalidStoreIdException();
    }
}
```

Fig6, 7: code after refactoring

We have chosen to extract the functionality of the method into different reusable methods: validateInput, validateOrderTime, calculatePrice, calculatePriceWithoutCoupons. The first one does input validation to check if there are no null parameters or negative numbers, etc. It also does checks on whether the user is allowed to place this order and whether the store exists. The second method does time validation by checking that the time selected for pickup is at least 30 minutes in the future. The third one calculates the price of an order when there are coupons inside, and the last one when there are no coupons. These are all separate actions and if any of them change logic we can just change the respective method.

Thereby we have reduced the LOC (max. 23) and reduced the CC (max. 9).

Second smell SendEmail*(…):* This method in the mailing service is also relatively long with a high cyclomatic complexity. It has 39 LOC, on the upper limit of acceptable LOC and a cyclomatic complexity of 5, which can be improved for this method (see figure below). This method currently finds the info of the address the message needs to be sent to, and then determines the contents of the message based on whether an order has been created, edited or deleted. In this case, extracting the method (switch statement) is a potential choice, but we chose to rewrite the method to make it more maintainable and reduce LOC and CC at the same time. Furthermore, we have seen that the use of switch cases is regarded as bad practice, so we decided to get rid of it.

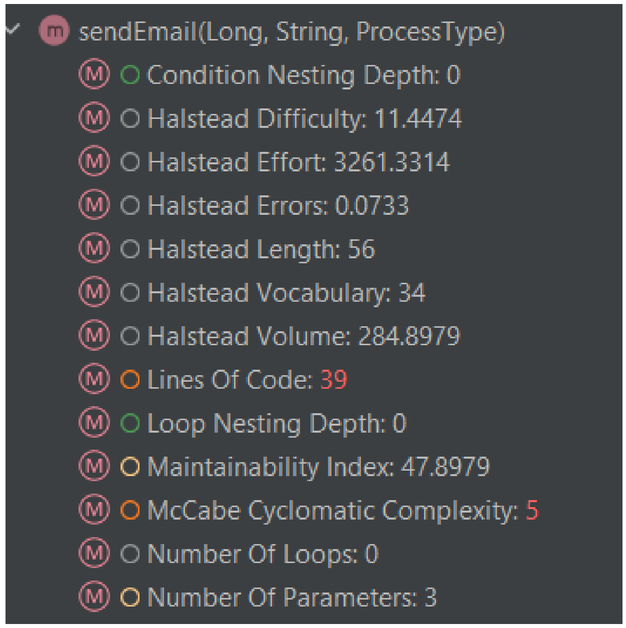


Fig 8: Code metrics of the sendEmail method before refactoring

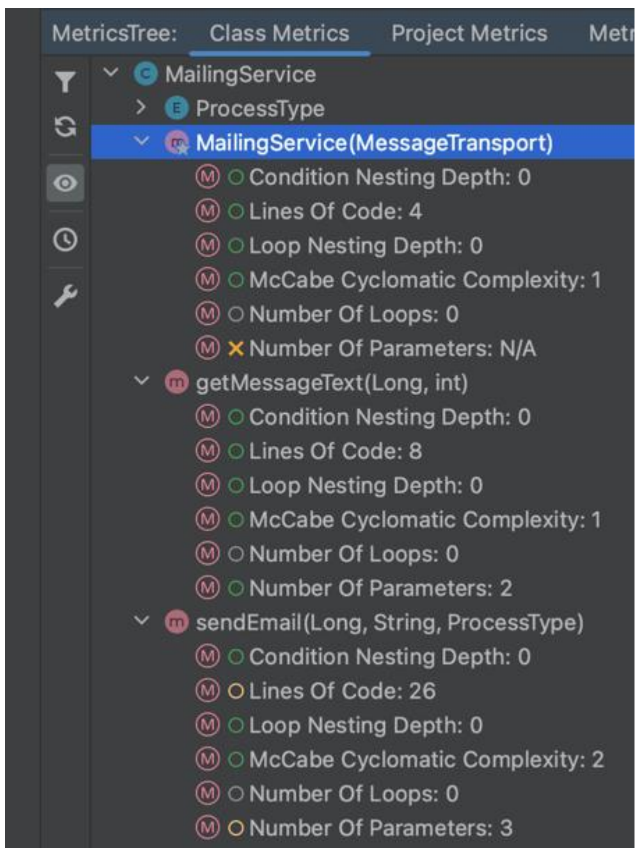


Fig 9: Code metrics after refactoring: extract message selection

```java
public void sendEmail(Long orderId, String recipientEmail, ProcessType processType) {
    try {
        // Create a default MimeMessage object.
        MimeMessage message = new MimeMessage(session);
        // Set From: header field of the header.
        message.setFrom(new InternetAddress(fromEmail));
        // Set To: header field of the header.
        message.addRecipient(Message.RecipientType.TO, new InternetAddress(recipientEmail));
        // Set Subject: header field
        switch (processType) {
            case EDITED:
                message.setSubject("Order has been edited!");
                // Now set the actual message
                message.setText(String.format("Order with orderId : %d has been edited", ord
                break;
            case CREATED:
                message.setSubject("Order has been created!");
                // Now set the actual message
                message.setText(String.format("Order with orderId : %d has been created", or
                break;
            case DELETED:
                message.setSubject("Order has been deleted!");
                // Now set the actual message
                message.setText(String.format("Order with orderId : %d has been deleted", or
                break;
        }
        // Send message
        messageTransport.sendMessage(message);
    } catch (MessagingException mex) {
        mex.printStackTrace();
    }
```

Fig 10: code before refactoring

RecipeService.java | MailingService.java | PriceControllerTests.java | AllergenController.java | StoreService.java | OrderService.java | IngredientService.java | RecipeController.ja

```java
3 usages
transient private Session session;
1 usage
private final transient String[] messageSubject = {
        "Order has been edited!",
        "Order has been created!",
        "Order has been deleted!"
};
no usages  Borislav Semerdzhiev
@Autowired
public MailingService(MessageTransport messageTransport) {
    this.messageTransport = messageTransport;

    String host = "smtp.gmail.com";

    // Get system properties
    Properties properties = System.getProperties();

    // Setup mail server
    properties.put("mail.smtp.host", host);
    properties.put("mail.smtp.port", "465");
    properties.put("mail.smtp.ssl.enable", "true");
    properties.put("mail.smtp.auth", "true");

    // Get the Session object.// and pass username and password
    session = Session.getInstance(properties, getPasswordAuthentication() → {
            return new PasswordAuthentication(fromEmail, fromPassword);
    });

    // Used to debug SMTP issues
    session.setDebug(true);
}

1 usage  Borislav Semerdzhiev
private String getMessageText(Long orderId, int index) {
    String[] messageText = {
            String.format("Order with orderId : %d has been edited", orderId),
            String.format("Order with orderId : %d has been created", orderId),
            String.format("Order with orderId : %d has been deleted", orderId)
    };
    return messageText[index];
}

/**
```

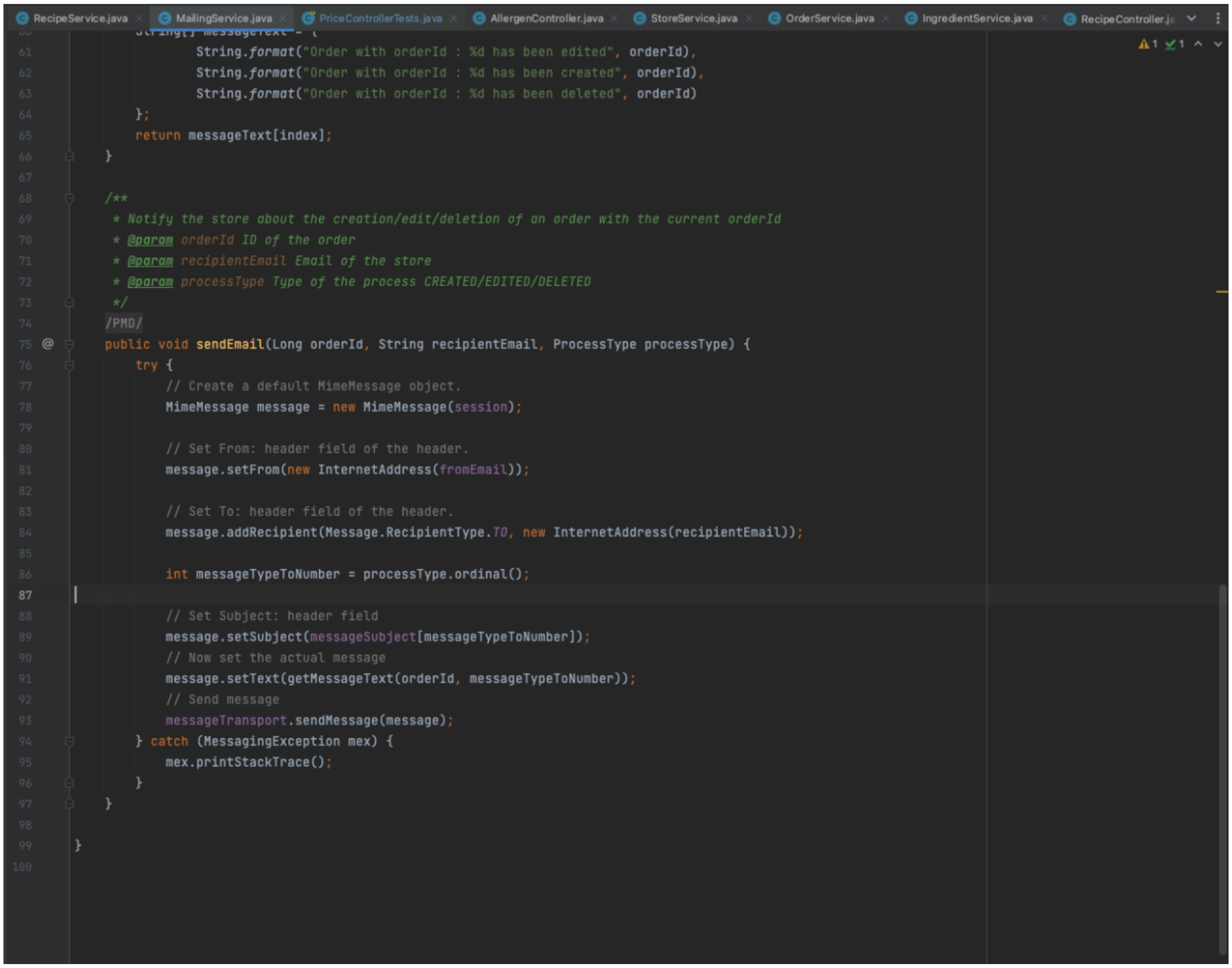


Fig 11, 12: code after refactoring

In order to extract the selection of a message body we have created an array that corresponds with the relation between the status of the order and the message to be sent. This becomes a class attribute. We also add a method getMessageText, which embeds the order id into the text to be put into the message. The original method now only takes care of choosing the right address and sending the email. The choice of removing the switch statement and now using an array makes the solution also more maintainable because instead of adding a full case to the switch, we can just add an element to the array.

This way, we have also reduced the LOC to max. 26 and CC to max. 2.

Third smell GetFoodPrices*(…)*: This method inside the food price service, being 45 lines of code long and having a cyclomatic complexity of 5 (see figure below), as well as being the only method in FoodPriceService, it made sense to refactor in order to improve code maintainability. This method made more sense to be split up into multiple methods with each performing individual tasks. Before refactoring, getFoodPrices() extracted ingredients ids and recipes ids from the order provided, sent a post request to get the prices for the ingredients and recipes, and it also extracted the response model from the response.

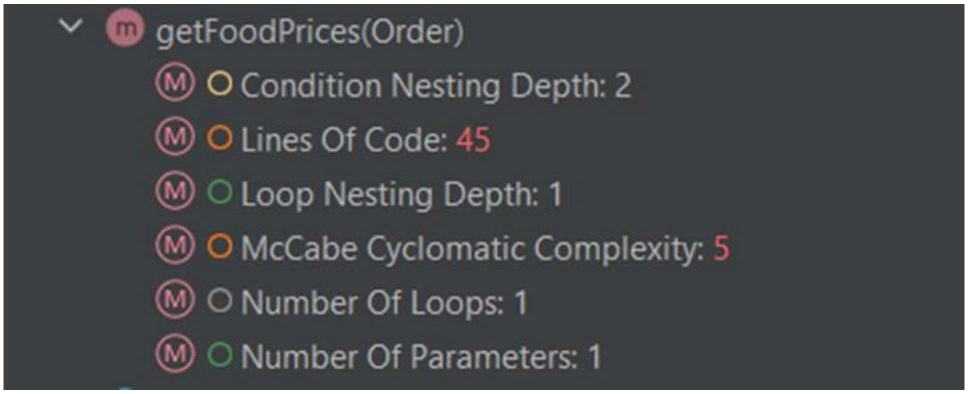


Fig 13: Code metrics before refactoring getFoodPrices

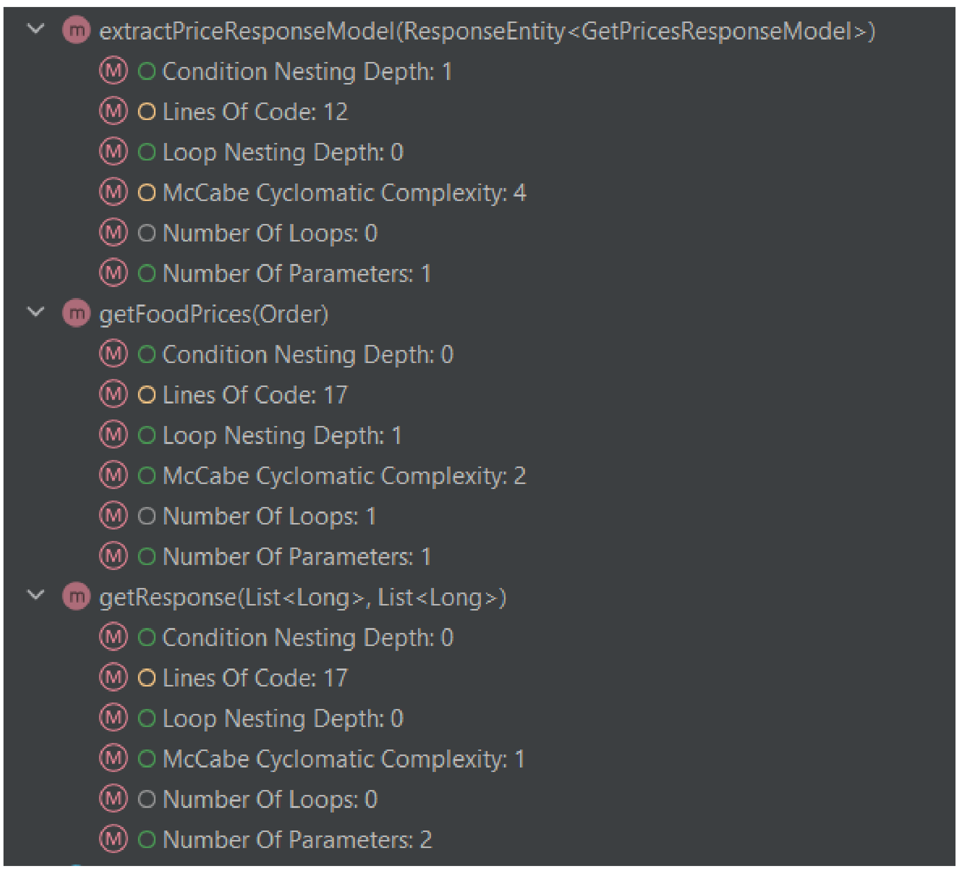


Fig 14: code metrics after refactoring

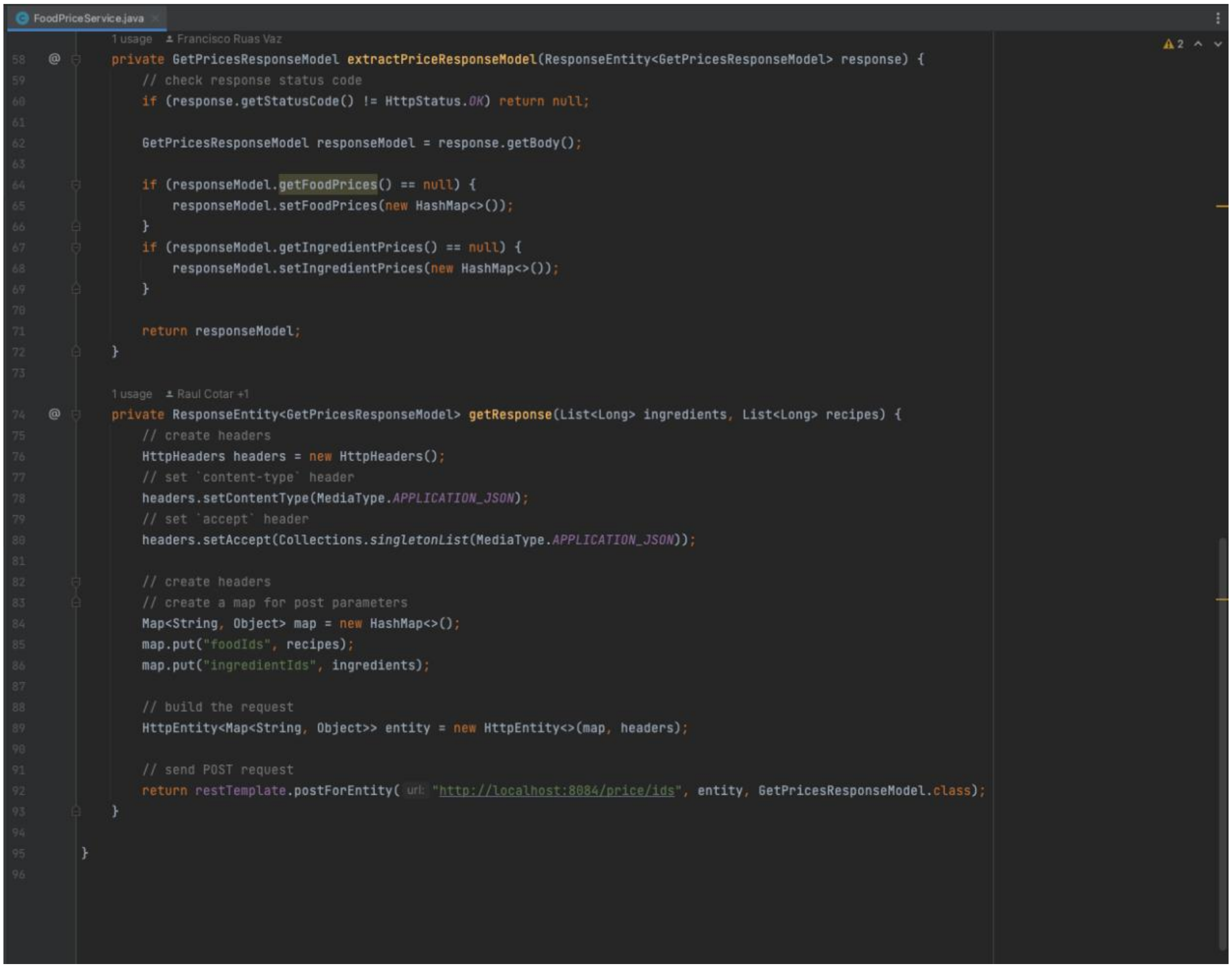
```java
public GetPricesResponseModel getFoodPrices(Order order) {
    List<Long> ingredients = new ArrayList<>();

    for (Food f: order.getFoods()) {
        ingredients.addAll(f.getExtraIngredients());
        ingredients.addAll(f.getBaseIngredients());
    }

    List<Long> recipes = order.getFoods().stream()
            .map(Food::getRecipeId).collect(Collectors.toList());

    // create headers
    HttpHeaders headers = new HttpHeaders();
    // set `content-type` header
    headers.setContentType(MediaType.APPLICATION_JSON);
    // set `accept` header
    headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));

    // create headers
    // create a map for post parameters
    Map<String, Object> map = new HashMap<>();
    map.put("foodIds", recipes);
    map.put("ingredientIds", ingredients);

    // build the request
    HttpEntity<Map<String, Object>> entity = new HttpEntity<>(map, headers);

    // send POST request
    ResponseEntity<GetPricesResponseModel> response =
            this.restTemplate.postForEntity( url: "http://localhost:8084/price/ids", entity, GetPricesResponseModel.class);
```

```java
    System.out.println(response);

    if (response.getStatusCode() == HttpStatus.OK) {
        GetPricesResponseModel responseModel = response.getBody();
        if (responseModel.getFoodPrices() == null) {
            responseModel.setFoodPrices(new HashMap<>());
        }
        if (responseModel.getIngredientPrices() == null) {
            responseModel.setIngredientPrices(new HashMap<>());
        }

        return responseModel;
    } else {
        return null;
    }
}
```

Fig 15, 16: code before refactoring

FoodPriceService.java

```
28 usages   Raul Cotar +2
public GetPricesResponseModel getFoodPrices(Order order) {
    List<Long> ingredients = new ArrayList<>();

    for (Food f: order.getFoods()) {
        ingredients.addAll(f.getExtraIngredients());
        ingredients.addAll(f.getBaseIngredients());
    }

    List<Long> recipes = order.getFoods().stream()
            .map(Food::getRecipeId).collect(Collectors.toList());

    ResponseEntity<GetPricesResponseModel> response = getResponse(ingredients, recipes);

    return extractPriceResponseModel(response);
}

1 usage   Francisco Ruas Vaz
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;
}

1 usage   Raul Cotar +1
private ResponseEntity<GetPricesResponseModel> getResponse(List<Long> ingredients, List<Long> recipes) {
    // create headers
    HttpHeaders headers = new HttpHeaders();
    // set `content-type` header
    headers.setContentType(MediaType.APPLICATION_JSON);
    // set `accept` header
    headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));

    // create headers
    // create a map for post parameters
```



```java
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;
}

private ResponseEntity<GetPricesResponseModel> getResponse(List<Long> ingredients, List<Long> recipes) {
    // create headers
    HttpHeaders headers = new HttpHeaders();
    // set `content-type` header
    headers.setContentType(MediaType.APPLICATION_JSON);
    // set `accept` header
    headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));

    // create headers
    // create a map for post parameters
    Map<String, Object> map = new HashMap<>();
    map.put("foodIds", recipes);
    map.put("ingredientIds", ingredients);

    // build the request
    HttpEntity<Map<String, Object>> entity = new HttpEntity<>(map, headers);

    // send POST request
    return restTemplate.postForEntity( url: "http://localhost:8084/price/ids", entity, GetPricesResponseModel.class);
}

}
```

Fig 17, 18: code after refactoring

Splitting up the tasks reduced the lines of code per method (max. 17) and the cyclomatic complexity (max. 4). This also means the readability of the code is improved while keeping the number of methods in the class at a reasonable number. One less significant improvement is the decrease in the condition nesting depth, which also improves readability.

We split the method into extracting the response model i.e., getting the prices response model from the response the food microservice sends (making null checks on it as well), getResponse (send the post request to the food microservice to get the prices) and the original method, which simply takes care of receiving data and sending the appropriate requests.

Fourth smell RecipeIsSafe*(…):* This method in the allergen service does not surpass the LOC limit. However, it does have a high cyclomatic complexity. If we consider that its purpose is simply to check a binary condition (whether the recipe is safe or not), it makes sense to refactor it to improve readability and maintainability. The recipe checks two things: if all the ingredients within the recipe exist, and if there is an ingredient the user is allergic to. We will split and rewrite the functionality of these methods into 2.

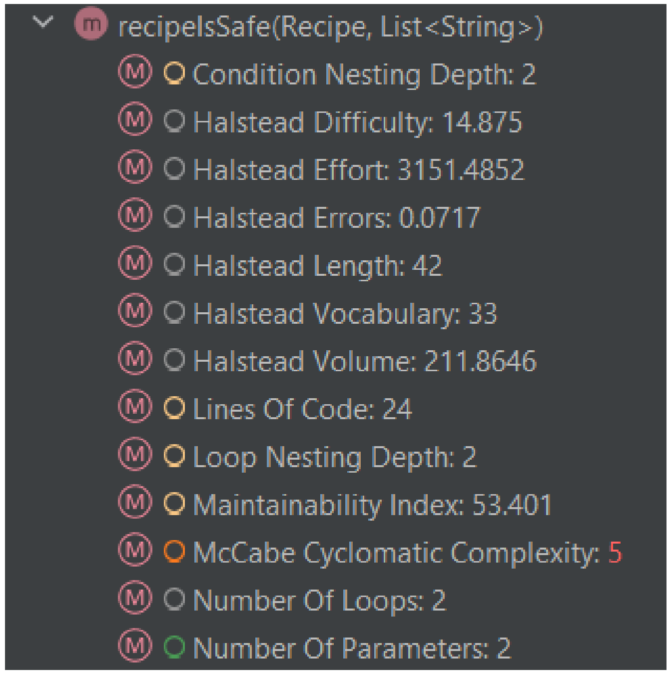


Fig 19: Code metrics before refactoring recipeIsSafe

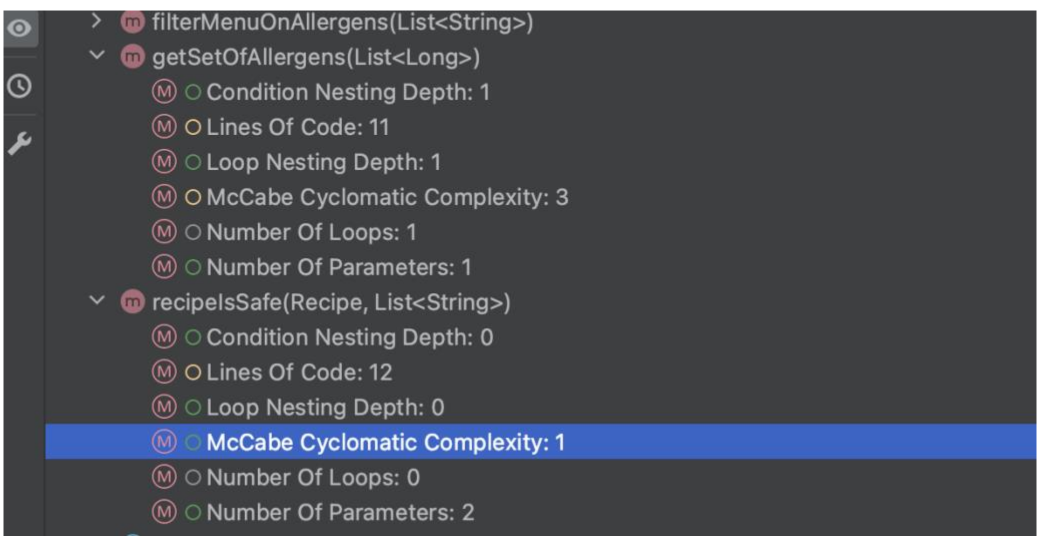


Fig 20: Code metrics after refactoring

```java
public boolean recipeIsSafe(Recipe recipe, List<String> allergens) throws IngredientNotFoundException {
    List<Long> ids = recipe.getBaseToppings();
    int size = ids.size();
    for (int i = 0; i < size; i++){
        long id = ids.get(i);
        if (ingredientRepository.existsById(id)){
            List<String> allergensOfIngredient = ingredientRepository.findById(id).get().getAllergens();
            for (String allergen: allergensOfIngredient){
                if (allergens.contains(allergen)) {
                    return false;
                }
            }

        } else {
            throw new IngredientNotFoundException();
        }
    }
    return true;
}
```

Fig 21: code before refactoring

```java
FoodPriceService.java    IngredientService.java    AllergenService.java
        }
        return menu;
    }


    /**
     * @param recipe Recipe we want to check for allergens
     * @param allergens list of strings that represents the allergens
     * @return true iff the recipe does not contain any of the allergens
     * @throws IngredientNotFoundException when an ingredient of this recipe is not stored in the database
     */
    16 usages  Borislav Semerdzhiev +2
    public boolean recipeIsSafe(Recipe recipe, List<String> allergens) throws IngredientNotFoundException {
        List<Long> ids = recipe.getBaseToppings();
        Set <String> recipeAllergens = getSetOfAllergens(ids);
        recipeAllergens.retainAll(new HashSet<>(allergens));
        return recipeAllergens.isEmpty();
    }

    /**
     * Returns a set of the allergens contained in the list of ids
     * @param ingredientIds list of ingredient ids
     * @return set of allergens
     * @throws IngredientNotFoundException
     */
    1 usage  Borislav Semerdzhiev +1
    @SuppressWarnings("PMD")
    private Set<String> getSetOfAllergens(List<Long> ingredientIds) throws IngredientNotFoundException {
        Set<String> allergens = new HashSet<>();

        for (Long ingredientId : ingredientIds) {
            if (!ingredientRepository.existsById(ingredientId)) {
                throw new IngredientNotFoundException();
            }
            allergens.addAll(ingredientRepository.findById(ingredientId).get().getAllergens());
        }
        return allergens;
    }

    /**
     * @param recipeId id of the recipe we want to check for allergens
     * @param allergens list of strings that represents the allergens
     * @return true iff the recipe does not contain any of the specified allergens
     * @throws RecipeNotFoundException when the recipe is not stored in the database
     * @throws IngredientNotFoundException when an ingredient of this recipe is not stored in the database
```

Fig 22: code after refactoring

We have now split it into 2 methods: getSetOfAllergens, which takes care of verifying the ingredients exist and collects the allergens of the recipe into a set. Now, in the main method, rather than looping over the allergens and checking for each if it is contained in the user allergen, we do set arithmetic to see if they have common items.

As a result, we reduce the cyclomatic complexity (max. 3, since we still need to check for existence of the ingredients) and a byproduct is reducing LOC to max. 12, which we can consider as positive since readability is better. We also have a lower loop nesting depth.

Fifth smell AddStore*(…):* This function in the store service takes care of adding a store into our database. Once more, for a simple purpose we have detected a high cyclomatic complexity, which is also partly due to verification on inputs and format. It takes care of verifying whether the input is null or the store already exists, and it also verifies the email and location formats. We can rewrite this method for less CC and as a byproduct reduce LOC.

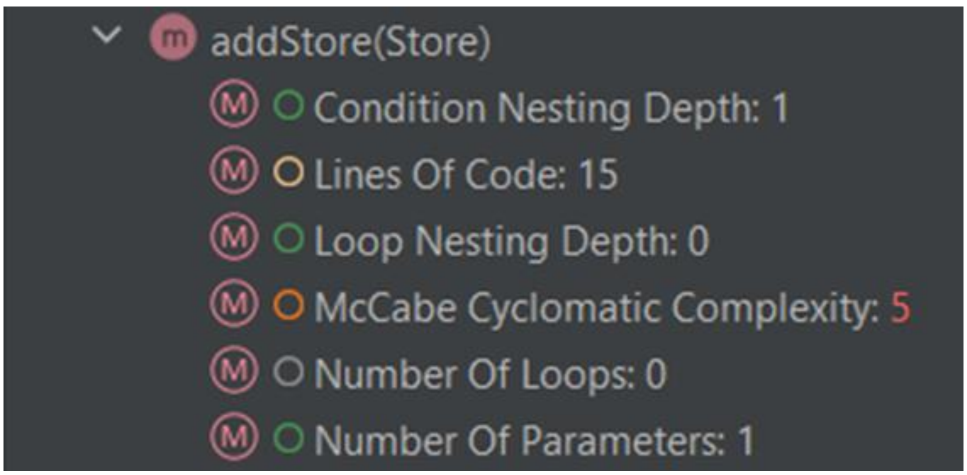


Fig 23: code metrics before refactoring the add store method

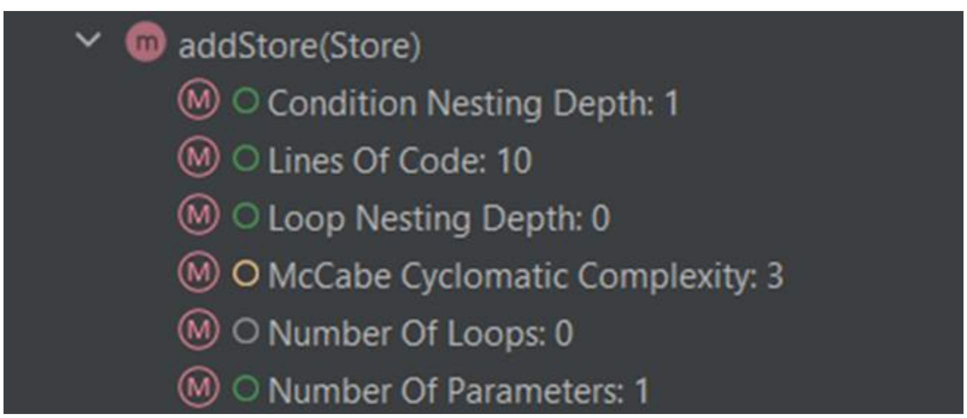


Fig 24: code metrics after refactoring

```java
public Store addStore(Store store) throws Exception {
    if (store == null) {
        throw new StoreIsNullException();
    }

    if (storeRepo.existsById(store.getId())) {
        throw new StoreAlreadyExistException();
    }

    if (!verifyEmailFormat(store.getContact())) {
        throw new InvalidEmailException();
    }

    if (!verifyLocationFormat(store.getLocation())) {
        throw new InvalidLocationException();
    }

    return storeRepo.save(store);
}
```

Fig 25: code before refactoring

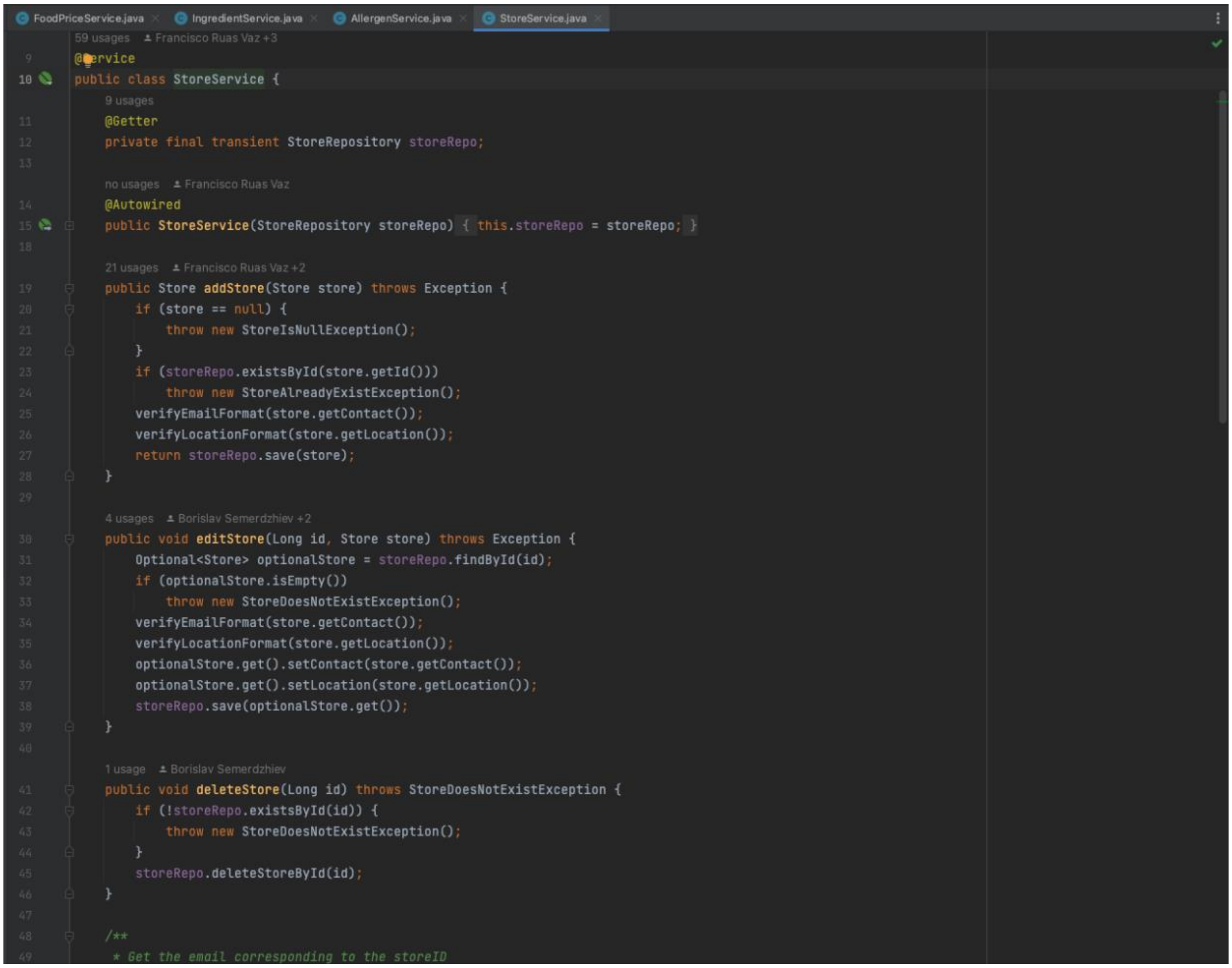


Fig 26: code after refactoring

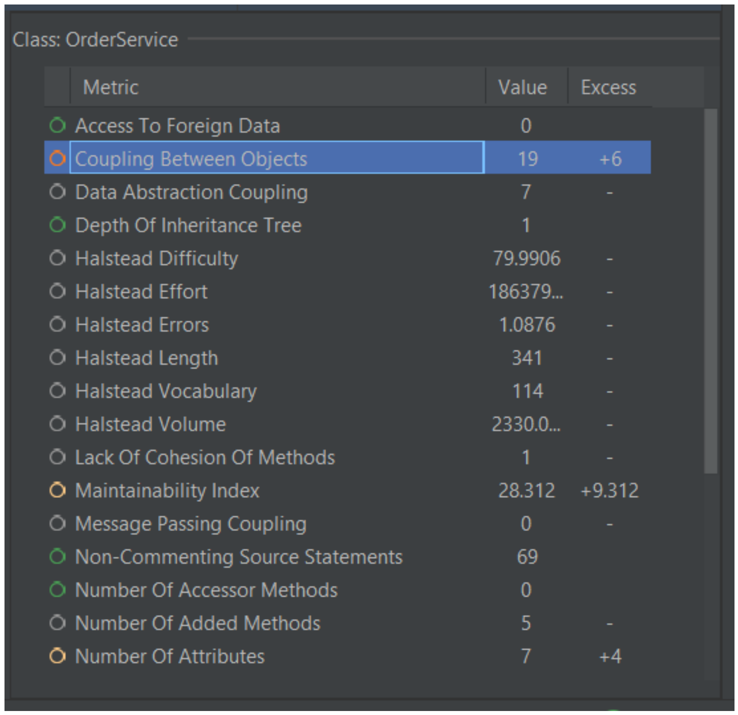
We did not extract any method in this case, but rather just rewrote things in a less complex manner. We had the methods that verify the formats throw their respective exceptions directly, so we only need to call them rather than making an additional if statement only to throw a condition. This reduces the CC to 3 and as a byproduct the LOC to 10.

Class smells:

First smell OrderService*:* The OrderService class is a significant part of the application and interacts with a lot of different services and repositories. This results in a high value of coupling between objects. This is demonstrated in the picture below where you can see a value of 19 between objects. For maintainability reasons this is undesirable and that's why we decide to change it.

After performing the method-level smell refactoring on the processOrder method inside OrderService we were left with multiple smaller methods, some of which performed a lot of operations that inherently belong to the Order class and required access to its fields. This was the main cause of our high coupling, because these methods should be performed in the order class as they inherently belong there and do not require any of the attributes of the OrderService class to work correctly. This led us to deciding to move the calculate price methods to the Order class which decreased the coupling tremendously.

Performing this change led to a lower number in the coupling between objects from 19 to 13 (demonstrated in the pictures below). Furthermore, we were able to bring the number of attributes down from 7 to 6 helping the maintainability.



Class: OrderService

Metric	Value	Excess
Access To Foreign Data	0	
Coupling Between Objects	19	+6
Data Abstraction Coupling	7	-
Depth Of Inheritance Tree	1	
Halstead Difficulty	79.9906	-
Halstead Effort	186379...	-
Halstead Errors	1.0876	-
Halstead Length	341	-
Halstead Vocabulary	114	-
Halstead Volume	2330.0...	-
Lack Of Cohesion Of Methods	1	-
Maintainability Index	28.312	+9.312
Message Passing Coupling	0	-
Non-Commenting Source Statements	69	
Number Of Accessor Methods	0	
Number Of Added Methods	5	-
Number Of Attributes	7	+4

Fig 27: metrics before refactoring

Class: OrderService

Metric	Value	Excess
Access To Foreign Data	1	
Coupling Between Objects	13	
Data Abstraction Coupling	6	-
Depth Of Inheritance Tree	1	
Lack Of Cohesion Of Methods	1	-
Message Passing Coupling	20	-
Non-Commenting Source Statements	38	
Number Of Accessor Methods	0	
Number Of Added Methods	6	-
Number Of Attributes	6	+3
Number Of Attributes And Methods	25	-
Number Of Children	0	
Number Of Methods	7	+1
Number Of Operations	19	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	24	
Tight Class Cohesion	0.6667	
Weight Of A Class	1.0	
Weighted Methods Per Class	15	+4

Fig 28: metrics after refactoring

```java
        || order.getPickupTime() == null || order.getCouponIds() == null)
        throw new 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 ord
    //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(
        //System.out.println(order.getUserId() + " " + orderRepo.findByOrderId(order.orderId));
        throw new InvalidEditException();
    }

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

    //check if the selected pickup time is 30 minutes or more in the future
    LocalDateTime current = clockWrapper.getNow();

    if (order.getPickupTime().isBefore(current.plusMinutes(30)))
        throw new TimeInvalidException();

    GetPricesResponseModel prices = foodPriceService.getFoodPrices(order); // get prices
    if (prices == null)
        //some food does not exist or something else went wrong in the food ms communication
        throw new FoodInvalidException();

    ArrayList<Coupon> coupons = new ArrayList<>(coupon_percentage_repository.findAllById(order.couponIds));
    coupons.addAll(coupon_2for1_repository.findAllById(order.couponIds));
    // this list only contains validated coupons, no need for additional checks
    order.couponIds.clear(); // clear the list, so we can send only the used one back
```

```java
    if (order.getPickupTime().isBefore(current.plusMinutes(30)))
        throw new TimeInvalidException();

    GetPricesResponseModel prices = foodPriceService.getFoodPrices(order); // get prices
    if (prices == null)
        //some food does not exist or something else went wrong in the food ms communication
        throw new FoodInvalidException();

    ArrayList<Coupon> coupons = new ArrayList<>(coupon_percentage_repository.findAllById(order.couponIds));
    coupons.addAll(coupon_2for1_repository.findAllById(order.couponIds));
    // this list only contains validated coupons, no need for additional checks
    order.couponIds.clear(); // clear the list, so we can send only the used one back

    //get the base price of the order
    double sum = 0.0;
    for (Food f: order.getFoods()) {
        sum += prices.getFoodPrices().get(f.getRecipeId()).getPrice();
        for (long l: f.getExtraIngredients()) {
            sum += prices.getIngredientPrices().get(l).getPrice();
        }
    }

    if (coupons.isEmpty()) { // If coupon list is empty, just add all ingredients and recipes
        final double EPS = 1e-6;
        if (Math.abs(order.price - sum) > EPS) {
            throw new PriceNotRightException("Price is not right");
        }
```

Fig 29,30: OrderService class(relevant part) before refactoring

```java
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);
}

/PMD/
private double calculatePriceWithCoupons(GetPricesResponseModel prices, List<Coupon> coupons, double sum) {
    if (coupons.isEmpty()) {
        return sum;
    }
    final double priceWithoutCoupons = sum;
    couponIds.add("0");

    for (Coupon c: coupons) {
        //iterate over the list of valid coupons
        double price = c.calculatePrice( order: this, prices, priceWithoutCoupons);

        if (Double.compare(price, sum) < 0) {
            sum = price;
            //set the first element in the coupon ids to the coupon used
            //order.couponIds.clear();
            couponIds.set(0, c.getId());
        }
    }
    return sum;
```

Fig 31: Order class after the refactoring

Second smell UserController*(...):* While inspecting the metrics of our tool, we saw that the UserController had intensive coupling with other objects. This was strange to us since the UserController does not implement a lot of functionality. After inspection we saw that the access to foreign data, coupling between objects was quite high for a class of that magnitude. That's why we decide to split up the controller into two different controllers.

We split the UserController up into the UserController (kept the same name) and the AllergenController since it was a logical division. In the original UserController the controller handled the requests for creating an account, logging in, and access to the user's allergens. The difference here is that creating an account and logging needs to communicate with the authentication microservice while the endpoints for accessing and updating the user's allergies does not require this.

We split the controllers into two separate controllers leading to a lower coupling between objects and access to foreign data. Additionally, doing this split will allow for further maintainability since this separates creating an account and logging from any updates and access to a user's account.

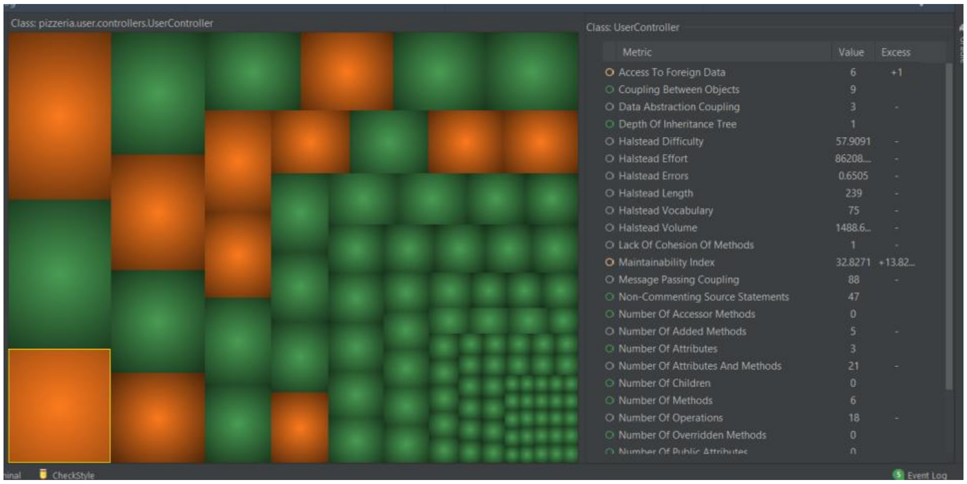


Fig 32: metrics before refactoring the UserController

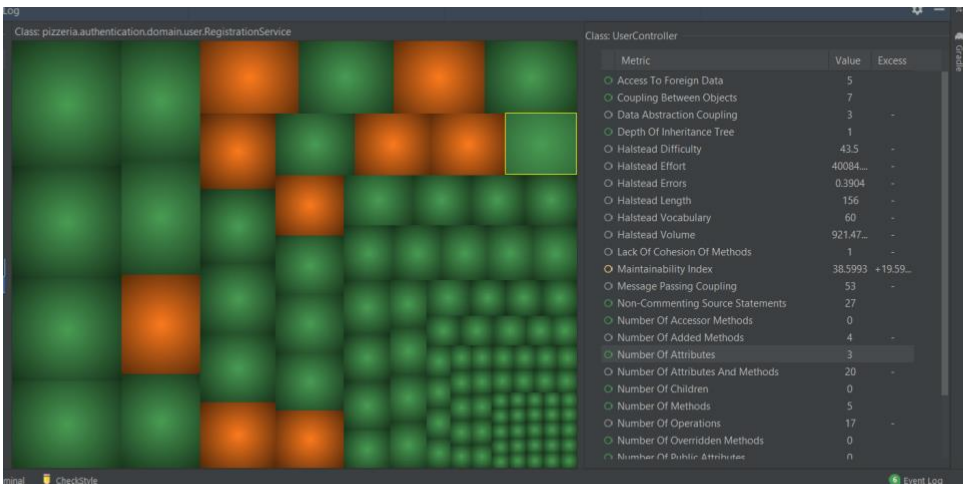
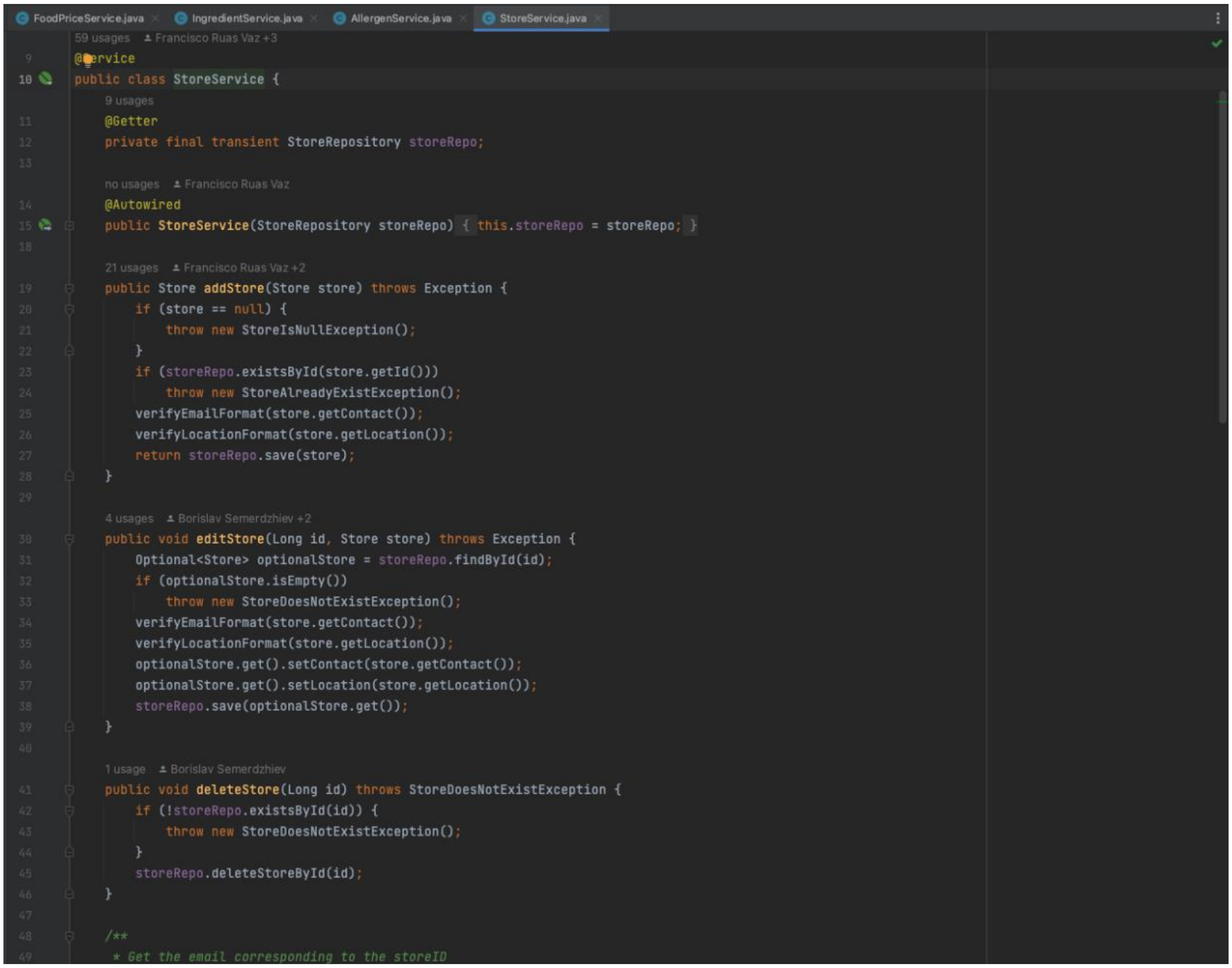
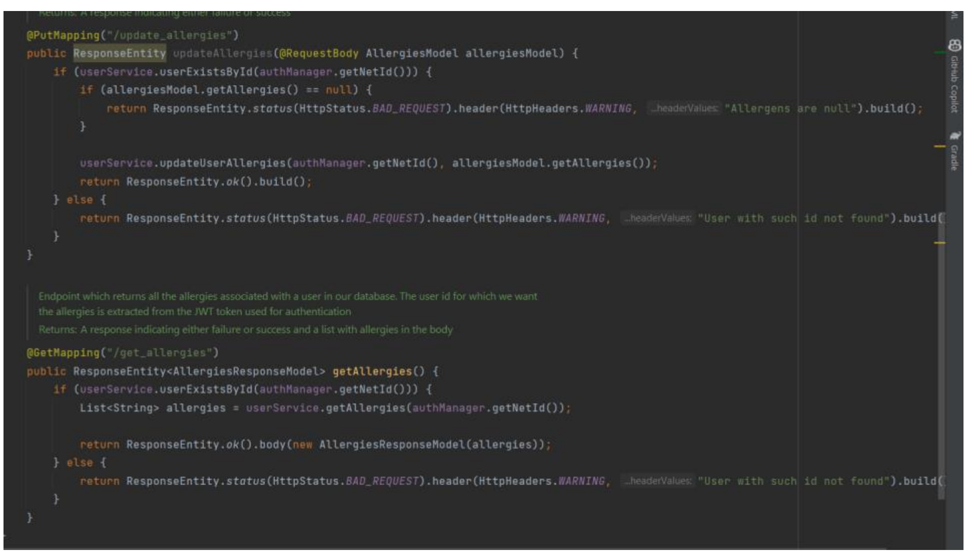


Fig 33: metrics after the refactoring of the UserController



```java
@Service
public class StoreService {
    @Getter
    private final transient StoreRepository storeRepo;

    @Autowired
    public StoreService(StoreRepository storeRepo) { this.storeRepo = storeRepo; }

    public Store addStore(Store store) throws Exception {
        if (store == null) {
            throw new StoreIsNullException();
        }
        if (storeRepo.existsById(store.getId()))
            throw new StoreAlreadyExistException();
        verifyEmailFormat(store.getContact());
        verifyLocationFormat(store.getLocation());
        return storeRepo.save(store);
    }

    public void editStore(Long id, Store store) throws Exception {
        Optional<Store> optionalStore = storeRepo.findById(id);
        if (optionalStore.isEmpty())
            throw new StoreDoesNotExistException();
        verifyEmailFormat(store.getContact());
        verifyLocationFormat(store.getLocation());
        optionalStore.get().setContact(store.getContact());
        optionalStore.get().setLocation(store.getLocation());
        storeRepo.save(optionalStore.get());
    }

    public void deleteStore(Long id) throws StoreDoesNotExistException {
        if (!storeRepo.existsById(id)) {
            throw new StoreDoesNotExistException();
        }
        storeRepo.deleteStoreById(id);
    }

    /**
     * Get the email corresponding to the storeID
```

Fig 34: Code before refactoring



```java
@PutMapping("/update_allergies")
public ResponseEntity updateAllergies(@RequestBody AllergiesModel allergiesModel) {
    if (userService.userExistsById(authManager.getNetId())) {
        if (allergiesModel.getAllergies() == null) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, "Allergens are null").build();
        }

        userService.updateUserAllergies(authManager.getNetId(), allergiesModel.getAllergies());
        return ResponseEntity.ok().build();
    } else {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, "User with such id not found").build(
    }
}

Endpoint which returns all the allergies associated with a user in our database. The user id for which we want
the allergies is extracted from the JWT token used for authentication
Returns: A response indicating either failure or success and a list with allergies in the body
@GetMapping("/get_allergies")
public ResponseEntity<AllergiesResponseModel> getAllergies() {
    if (userService.userExistsById(authManager.getNetId())) {
        List<String> allergies = userService.getAllergies(authManager.getNetId());

        return ResponseEntity.ok().body(new AllergiesResponseModel(allergies));
    } else {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, "User with such id not found").build(
    }
}
```

Fig 35: code extracted from the UserController that was moved to the AllergiesController after refactoring

Third smell AllergenController(…): We believe that the controllers should not do any computations, making any requests or having any concrete logic. They should only be used to route the request to the correct service. Upon inspection of the AllergenController, we saw that the controller was making multiple requests to the user microservice and implementing some logic. This led to quite a high coupling between objects (demonstrated in the metrics: 7) in the AllergenController which we do not want in a controller.

Therefore, we decided to move all the logic of the AllergenController to the AllergenService. This led to a reduced coupling between objects to 4 as demonstrated in the picture below. Furthermore, we decided to only catch an Exception instead of all the different types of exceptions we have. The reason for this is that we only pass the message and not the type of exception. This decreased maintainability.

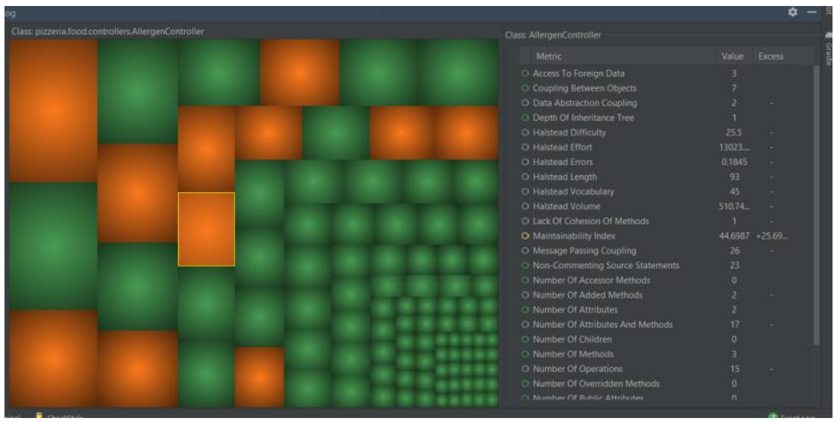


Fig 36: metrics before the refactoring

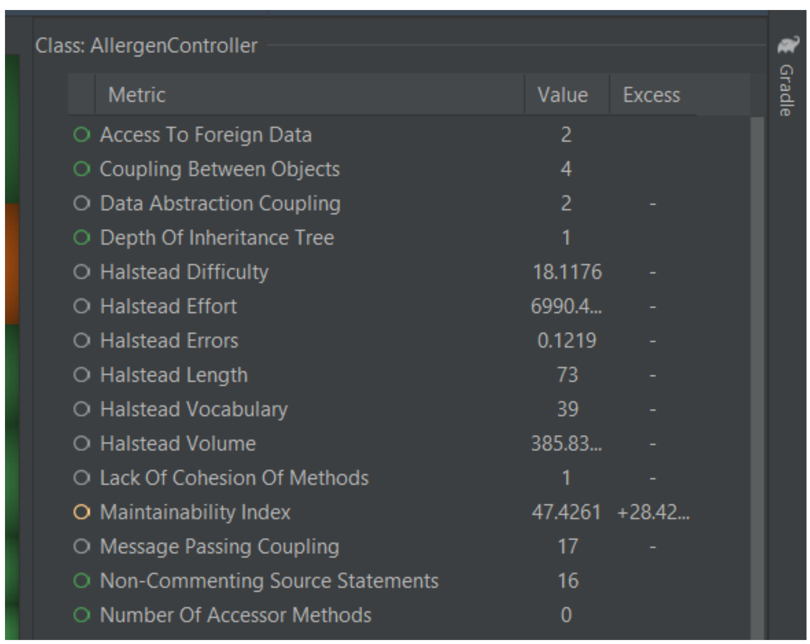


Fig 37: metrics after the refactoring

```java
@GetMapping("/menu")
public ResponseEntity<FilterMenuResponseModel> filterMenu(@RequestHeader(HttpHeaders.AUTHORIZATION) String token) {

        Optional<List<String>> allergens = requestService.getUserAllergens(token);
        if (allergens.isPresent()) {
            try {
                List<Recipe> filteredMenu = allergenService.filterMenuOnAllergens(allergens.get());
                FilterMenuResponseModel responseModel = new FilterMenuResponseModel();
                responseModel.setRecipes(filteredMenu);
                return ResponseEntity.status(HttpStatus.OK).body(responseModel);
            } catch (IngredientNotFoundException e) {
                return ResponseEntity.badRequest().header(HttpHeaders.WARNING, e.getMessage()).build();
            }
        } else {
            return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
        }

}

@GetMapping("/warn")
public ResponseEntity<Boolean> checkIfSafe(@RequestHeader(HttpHeaders.AUTHORIZATION) String token, @RequestBody Chec
    Optional<List<String>> allergens = requestService.getUserAllergens(token);
    if (allergens.isPresent()) {
        try {
            boolean isSafe = allergenService.checkIfSafeRecipeWithId(requestModel.getId(), allergens.get());
            return ResponseEntity.status(HttpStatus.OK).body(isSafe);
        } catch (RecipeNotFoundException e) {
            return ResponseEntity.badRequest().header(HttpHeaders.WARNING, e.getMessage()).build();
        } catch (IngredientNotFoundException e) {
            return ResponseEntity.badRequest().header(HttpHeaders.WARNING, e.getMessage()).build();
        }
```

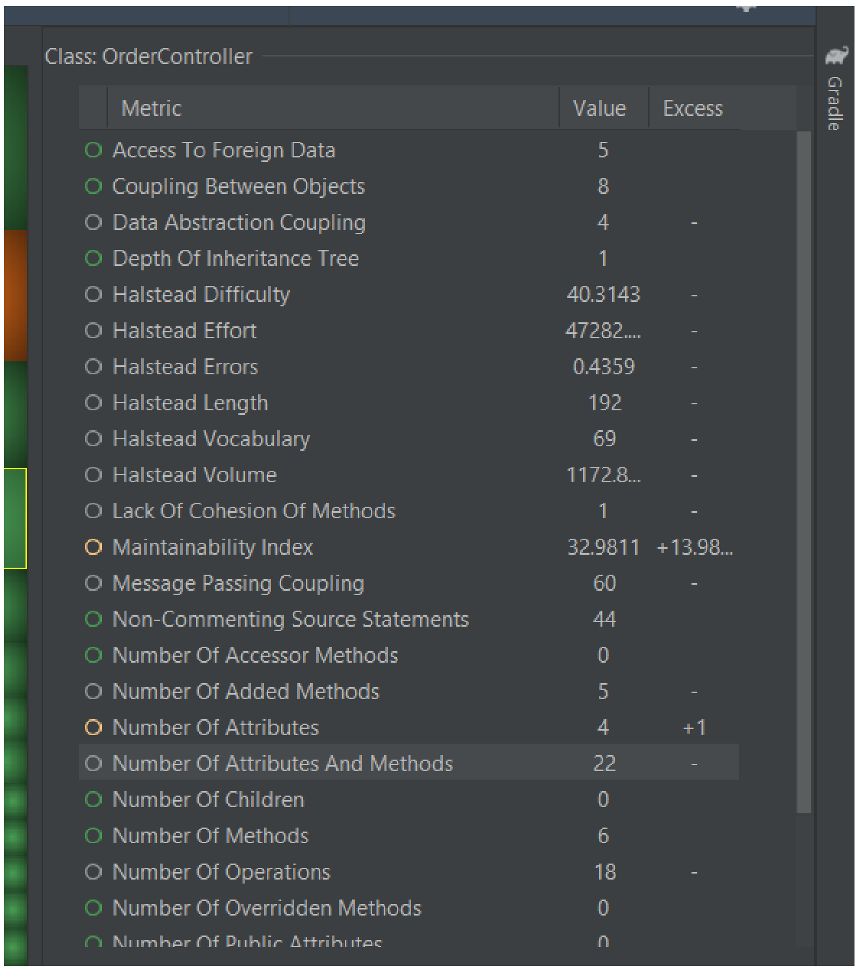
Fig 38: code before refactoring

```java
    }


    /**
     * @param token The token of the user
     * @return List of recipes that don't contain the specified allergens
     */
    3 usages  Borislav Semerdzhiev +1
    @GetMapping("/menu")
    public ResponseEntity<FilterMenuResponseModel> filterMenu(@RequestHeader(HttpHeaders.AUTHORIZATION) String token) {
        try {
            FilterMenuResponseModel responseModel = allergenService.filterMenu(token);
            if (responseModel == null) {
                return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
            }
            return ResponseEntity.status(HttpStatus.OK).body(allergenService.filterMenu(token));
        } catch (Exception e) {
            return ResponseEntity.badRequest().header(HttpHeaders.WARNING, e.getMessage()).build();
        }
    }

    5 usages  Borislav Semerdzhiev +1 *
    @GetMapping("/warn")
    public ResponseEntity<Boolean> checkIfSafe(@RequestHeader(HttpHeaders.AUTHORIZATION) String token, @RequestBody CheckIfRecipe
        try {
            Optional<Boolean> checkSafetyStatus = allergenService.checkSafety(token, requestModel);
            if (checkSafetyStatus.isEmpty()) {

                return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
            }
            return ResponseEntity.status(HttpStatus.OK).body(checkSafetyStatus.get());

        } catch (Exception e) {
            return ResponseEntity.badRequest().header(HttpHeaders.WARNING, e.getMessage()).build();
        }
    }
}
```

Fig 39: code after the refactoring

Fourth smell OrderController*:* As mentioned above we believe that a controller should not implement any functionality except for passing the request to the appropriate service and method. That is why we decided to refactor the order controller. We saw the order controller was performing input validation, sending emails and even performing some order logic. This led to a high coupling between the objects for a controller (demonstrated in the picture below: 8) and a very high message passing coupling. This demonstrates that the controller is implementing a lot of logic.

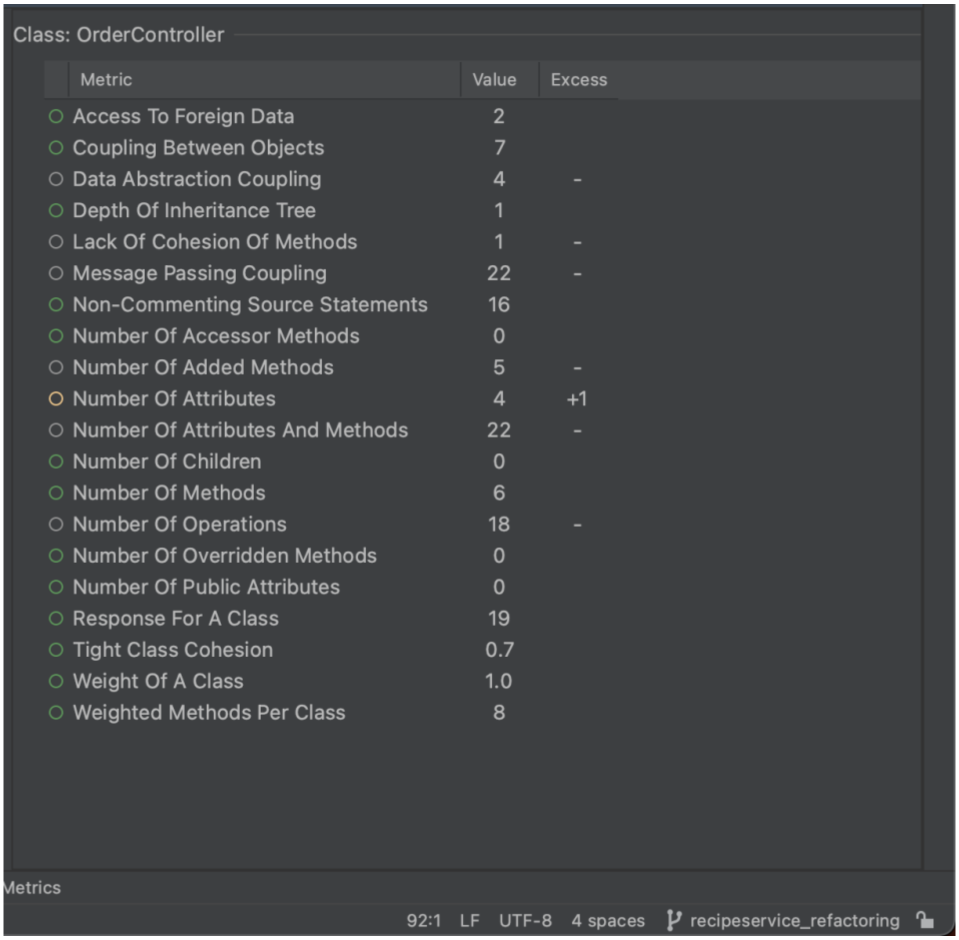
We decided to move all this logic to the OrderService such that the controller is only responsible for passing on the data to the service. This led to a decrease in access to foreign data and a decrease in coupling between objects as demonstrated in the pictures below. The biggest decrease can be found in the message passing coupling that went down from 60 to 22.



Class: OrderController

Metric	Value	Excess
Access To Foreign Data	5	
Coupling Between Objects	8	
Data Abstraction Coupling	4	-
Depth Of Inheritance Tree	1	
Halstead Difficulty	40.3143	-
Halstead Effort	47282....	-
Halstead Errors	0.4359	-
Halstead Length	192	-
Halstead Vocabulary	69	-
Halstead Volume	1172.8...	-
Lack Of Cohesion Of Methods	1	-
Maintainability Index	32.9811	+13.98...
Message Passing Coupling	60	-
Non-Commenting Source Statements	44	
Number Of Accessor Methods	0	
Number Of Added Methods	5	-
Number Of Attributes	4	+1
Number Of Attributes And Methods	22	-
Number Of Children	0	
Number Of Methods	6	
Number Of Operations	18	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	

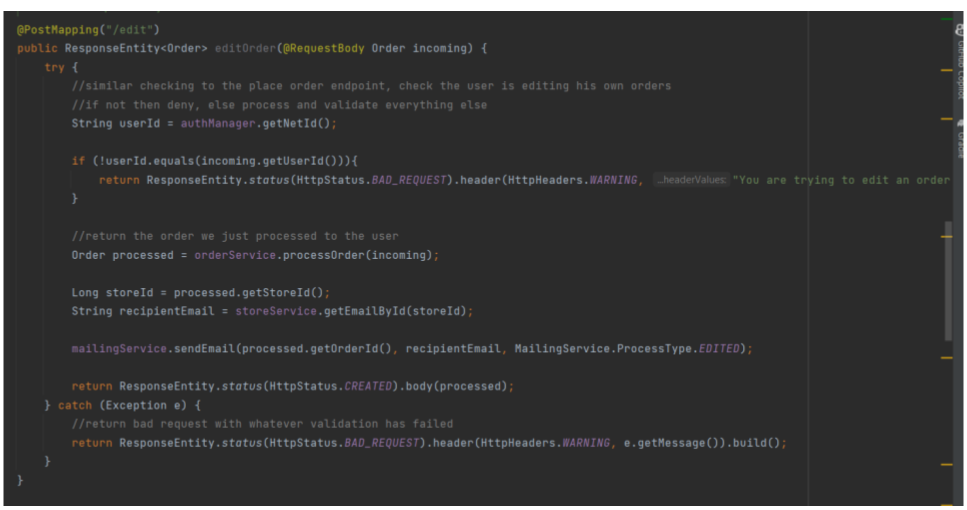
Fig 40: metrics before the refactoring



Class: OrderController

Metric	Value	Excess
Access To Foreign Data	2	
Coupling Between Objects	7	
Data Abstraction Coupling	4	-
Depth Of Inheritance Tree	1	
Lack Of Cohesion Of Methods	1	-
Message Passing Coupling	22	-
Non-Commenting Source Statements	16	
Number Of Accessor Methods	0	
Number Of Added Methods	5	-
Number Of Attributes	4	+1
Number Of Attributes And Methods	22	-
Number Of Children	0	
Number Of Methods	6	
Number Of Operations	18	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	19	
Tight Class Cohesion	0.7	
Weight Of A Class	1.0	
Weighted Methods Per Class	8	

Fig 41: metrics after the refactoring



```java
@PostMapping("/edit")
public ResponseEntity<Order> editOrder(@RequestBody Order incoming) {
    try {
        //similar checking to the place order endpoint, check the user is editing his own orders
        //if not then deny, else process and validate everything else
        String userId = authManager.getNetId();

        if (!userId.equals(incoming.getUserId())){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, "You are trying to edit an order
        }

        //return the order we just processed to the user
        Order processed = orderService.processOrder(incoming);

        Long storeId = processed.getStoreId();
        String recipientEmail = storeService.getEmailById(storeId);

        mailingService.sendEmail(processed.getOrderId(), recipientEmail, MailingService.ProcessType.EDITED);

        return ResponseEntity.status(HttpStatus.CREATED).body(processed);
    } catch (Exception e) {
        //return bad request with whatever validation has failed
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, e.getMessage()).build();
    }
}
```

```java
@DeleteMapping("/delete")
/PMD/
public ResponseEntity<Order> deleteOrder(@RequestBody DeleteModel deleteModel) {
    //get the user that is trying to delete the order
    String userId = authManager.getNetId();
    //check if the user is a manager
    boolean isManager = authManager.getRole().equals("[ROLE_MANAGER]");

    Optional <Order> orderToBeDeleted = orderService.findOrder(deleteModel.getOrderId());

    if (orderToBeDeleted.isPresent()) {
        Long storeId = orderToBeDeleted.get().getStoreId();
        String recipientEmail = storeService.getEmailById(storeId);

        if (!orderService.removeOrder(deleteModel.getOrderId(), userId, isManager)) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        }

        mailingService.sendEmail(deleteModel.getOrderId(), recipientEmail, MailingService.ProcessType.DELETED);
        //validate if we can delete this order, if we can ok else bad request
        return ResponseEntity.status(HttpStatus.OK).build();
    }

    return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
}
```

Fig 42, 43: code before the refactoring

FoodPriceService.java   IngredientService.java   AllergenService.java   StoreService.java   UserController.java   OrderController.java

```java
@PostMapping("/place")
public ResponseEntity<Order> placeOrder(@RequestBody Order incoming) {
    try {
        return orderOperationService.placeOrder(incoming, authManager.getNetId());
    } catch (Exception e) {
        //return bad request with whatever validation has failed
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, e.getMessage()).build();
    }
}

/**
 * Edit an order endpoint, updates the order in the database if valid
 * Includes validation of user and processes order in order service
 *
 * @param incoming the incoming order
 * @return the response entity
 */
no usages   Raul Cotar +2
@PostMapping("/edit")
public ResponseEntity<Order> editOrder(@RequestBody Order incoming) {
    try {
        return orderOperationService.editOrder(incoming, authManager.getNetId());
    } catch (Exception e) {
        //return bad request with whatever validation has failed
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).header(HttpHeaders.WARNING, e.getMessage()).build();
    }
}

/**
 * Delete order endpoint, deletes from the database if valid request
 * Includes user validation and processes order in order service
 *
 * @param deleteModel model containing the order id we want to remove
 * @return the response entity
 */
no usages   Borislav Semerdzhiev +2
@DeleteMapping("/delete")
/PMD/
public ResponseEntity<Order> deleteOrder(@RequestBody DeleteModel deleteModel) {
    return orderOperationService.deleteOrder(deleteModel, authManager.getNetId(), authManager.getRole());
}

/**
 * List orders endpoint, lists all the orders belonging to a user
```

Fig 44: code after refactoring

Fifth smell RecipeService: Using the metrics we identified the RecipeService class as containing too many methods and after further inspections, we confirmed that:

- It contained a method which should have been inside the IngredientService
- This class should be split in two, as it also makes more sense logically because of the nature of the methods inside

Once again, the metrics fully support our choice here (Weighted Methods Per Class had an excess of +15 pre-refactoring).

After the refactoring had been done we moved *checkForIngredientsExistence()* to the IngredientService and split the *RecipeService* leaving *registerFood(), updateFood(), deleteFood()* in the original class, but moving getPrices(), getMenu(), getBaseToppings() in the new class ***RecipeServiceResponseInformation***. This class division made sense to us since all the aforementioned methods had to do with retrieving information.

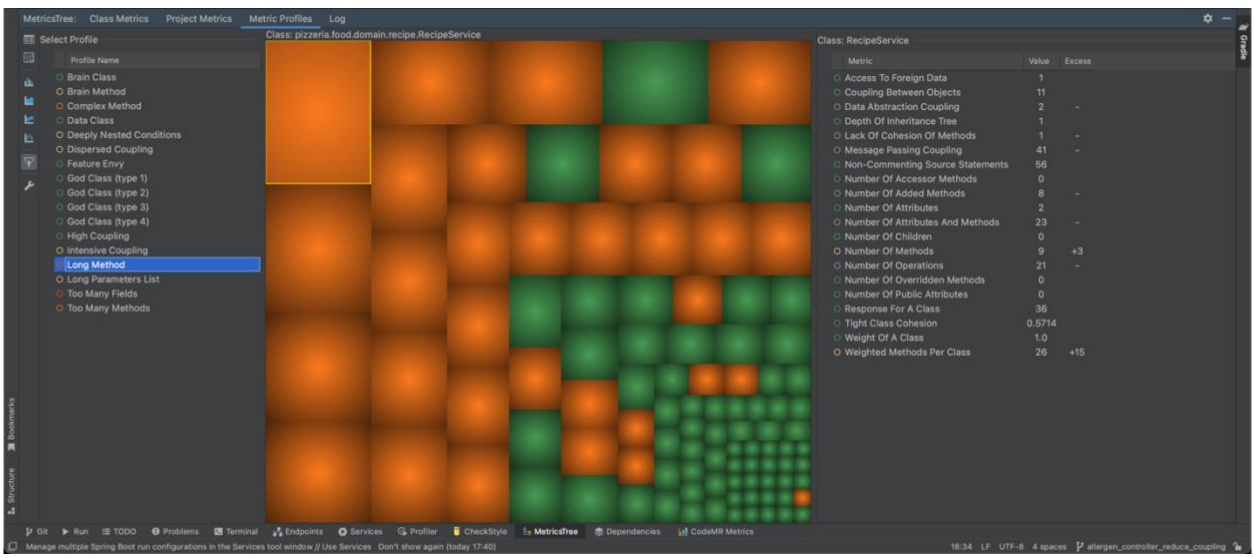
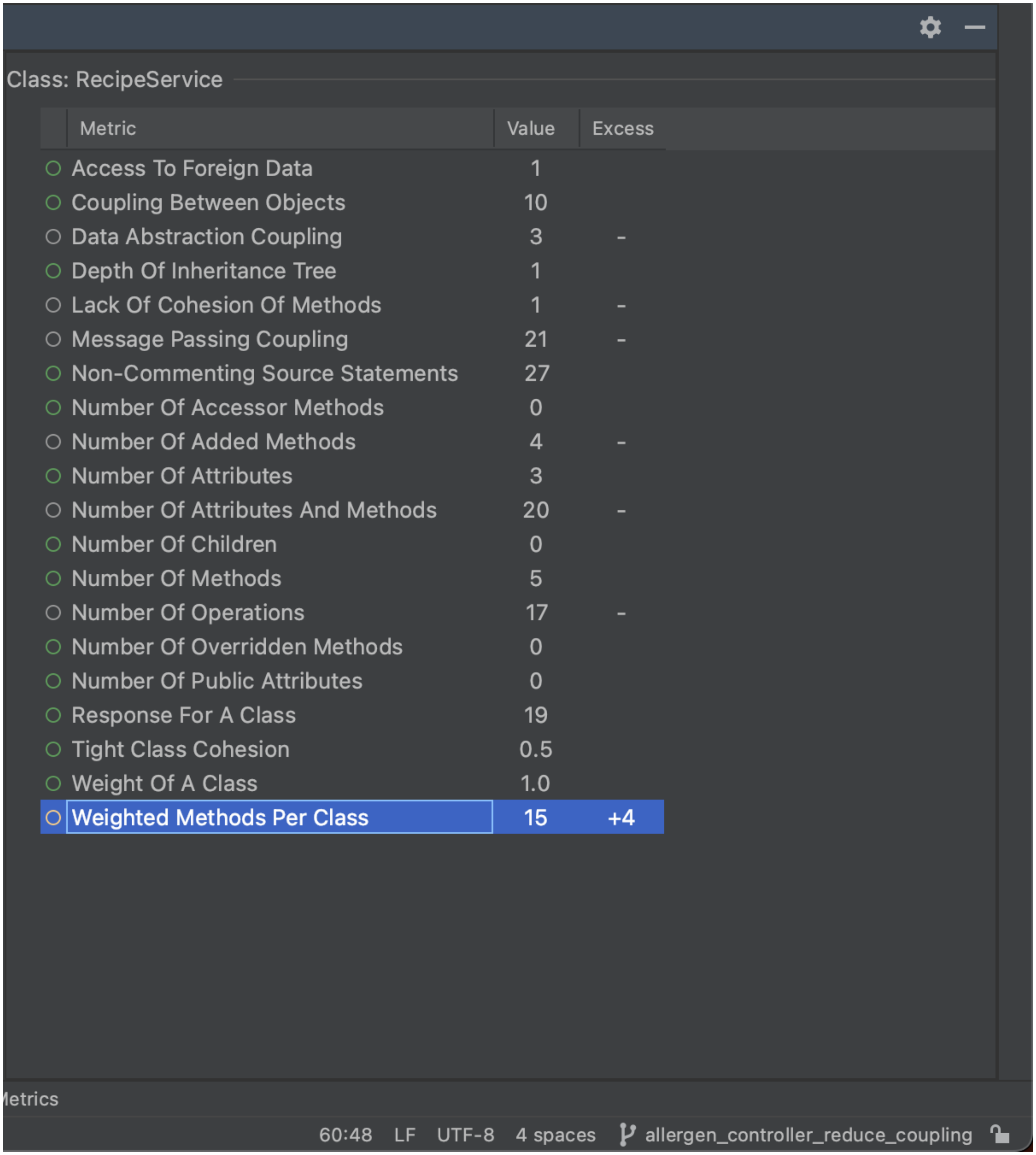


Fig 45: Metrics before the refactoring



Metric	Value	Excess
Access To Foreign Data	1	
Coupling Between Objects	10	
Data Abstraction Coupling	3	-
Depth Of Inheritance Tree	1	
Lack Of Cohesion Of Methods	1	-
Message Passing Coupling	21	-
Non-Commenting Source Statements	27	
Number Of Accessor Methods	0	
Number Of Added Methods	4	-
Number Of Attributes	3	
Number Of Attributes And Methods	20	-
Number Of Children	0	
Number Of Methods	5	
Number Of Operations	17	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	19	
Tight Class Cohesion	0.5	
Weight Of A Class	1.0	
Weighted Methods Per Class	15	+4

Fig 46: Metrics after the refactoring (huge improve for Weighted Methods Per Class and Number of Methods)

```java
public Recipe registerFood(Recipe recipe) throws RecipeAlreadyInUseException, IngredientNotFoundException, InvalidRecipeException {
    if (!userInputValidation(recipe)){
        throw new InvalidRecipeException();
    }
    if (recipeRepository.existsById(recipe.getId()) || recipeRepository.existsByName(recipe.getName())) {
        throw new RecipeAlreadyInUseException();
    }
    for (long id: recipe.getBaseToppings()){
        if (!ingredientRepository.existsById(id)){
            throw new IngredientNotFoundException();
        }
    }
    Recipe result = recipeRepository.save(recipe);
    return result;
}

/**
 * @param recipe Recipe instance that carries the data that we want to store instead of the recipe
 *               that is currently in the database.
 * @param id long value representing the id of the recipe we want to update.
 * @return the Recipe that is saved in the database if we are able to update it
 * @throws RecipeNotFoundException thrown when the given id is not associated with a recipe in the
 * database.
 * @throws IngredientNotFoundException thrown when one of the ingredient ids in the recipe is not stored in the
 * database.
 */
public Recipe updateFood(Recipe recipe, long id) throws RecipeNotFoundException, IngredientNotFoundException, InvalidRecipeException {
    if (!userInputValidation(recipe)){
        throw new InvalidRecipeException();
    }
    for (long idIngredients: recipe.getBaseToppings()){
        if (!ingredientRepository.existsById(idIngredients)){
            throw new IngredientNotFoundException();
        }
    }
    if (recipeRepository.existsById(id)) {
        recipe.setId(id);
        return recipeRepository.save(recipe);
    }
    throw new RecipeNotFoundException();
}

/**
```

RecipeService.java | PriceControllerTests.java | AllergenController.java | StoreService.java | OrderService.java | IngredientService.java | RecipeController.java | HttpRequestSer

```java
    public boolean deleteFood(long id) throws RecipeNotFoundException {
        if (recipeRepository.existsById(id)) {
            recipeRepository.deleteById(id);
            return true;
        }
        throw new RecipeNotFoundException();
    }

    /**
     * @param ids List longs representing the ids of the recipes of which we want to get the prices
     * @return a List of doubles that are the prices of the recipes
     * @throws RecipeNotFoundException thrown when one of the ids of the recipes was not in the database.
     */
    12 usages  Laurens Michielsen +1
    @SuppressWarnings("PMD")
    public Map<Long, Tuple> getPrices(List<Long> ids) throws RecipeNotFoundException {
        if (ids == null) {
            return new HashMap<>();
        }

        Map<Long, Tuple> prices = new HashMap<>(ids.size());
        for (long id: ids){
            if (recipeRepository.existsById(id)) {
                Recipe recipe = recipeRepository.findById(id).get();
                prices.put(id, new Tuple(recipe.getBasePrice(), recipe.getName()));
            } else {
                throw new RecipeNotFoundException("The Recipe with the id " + id + " was not found in the databases");
            }
        }
        return prices;
    }

    /**
     * @return List of recipes that represents the menu
     */
    4 usages  Laurens Michielsen
    public List<Recipe> getMenu(){
        return recipeRepository.findAll();
    }

    /**
     * given a recipe id return the associated ingredients
     * @param id long value representing the id of the recipe
     * @return List of Ingredients representing the basetoppings that are fetched from the ingredientRepository
```

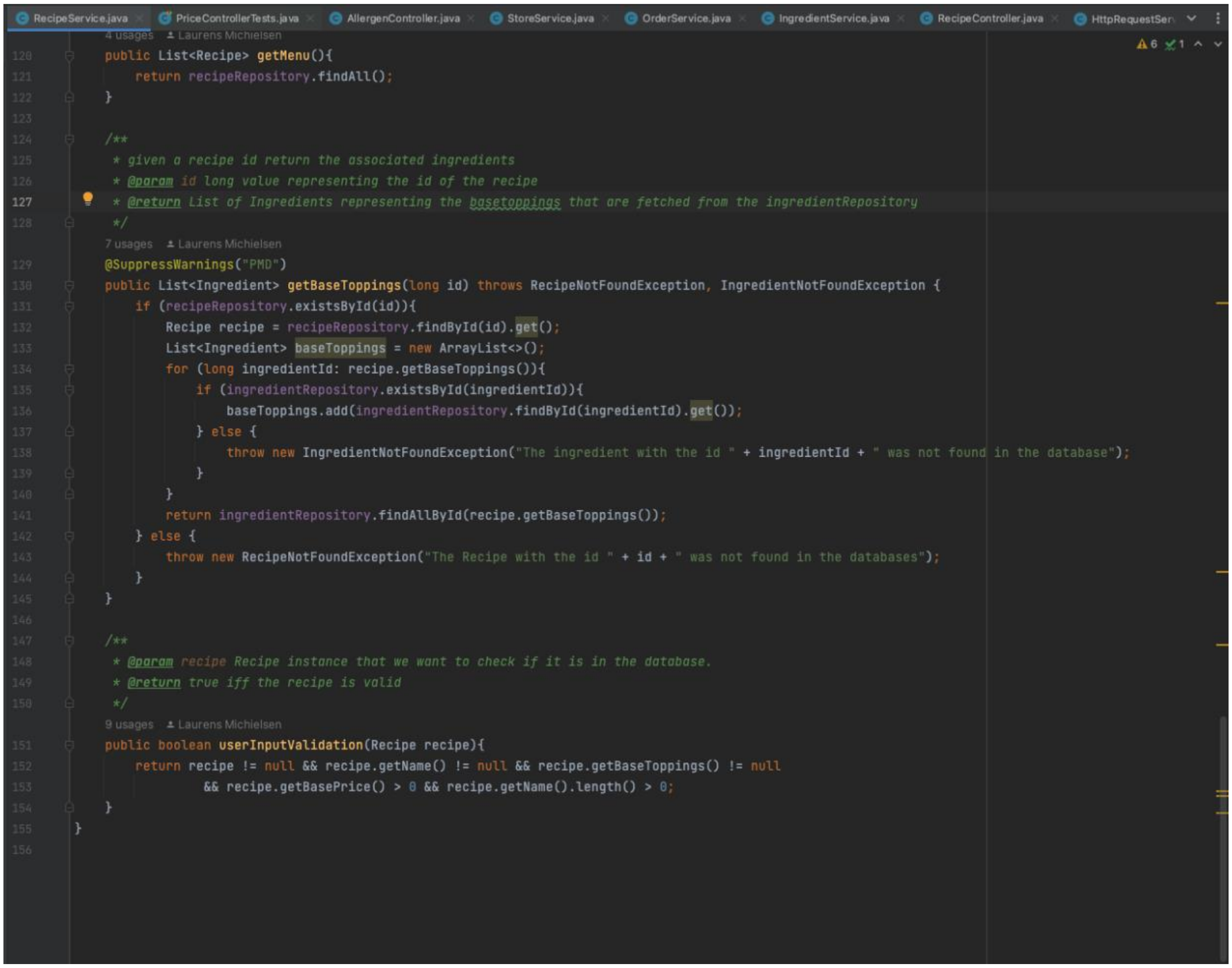
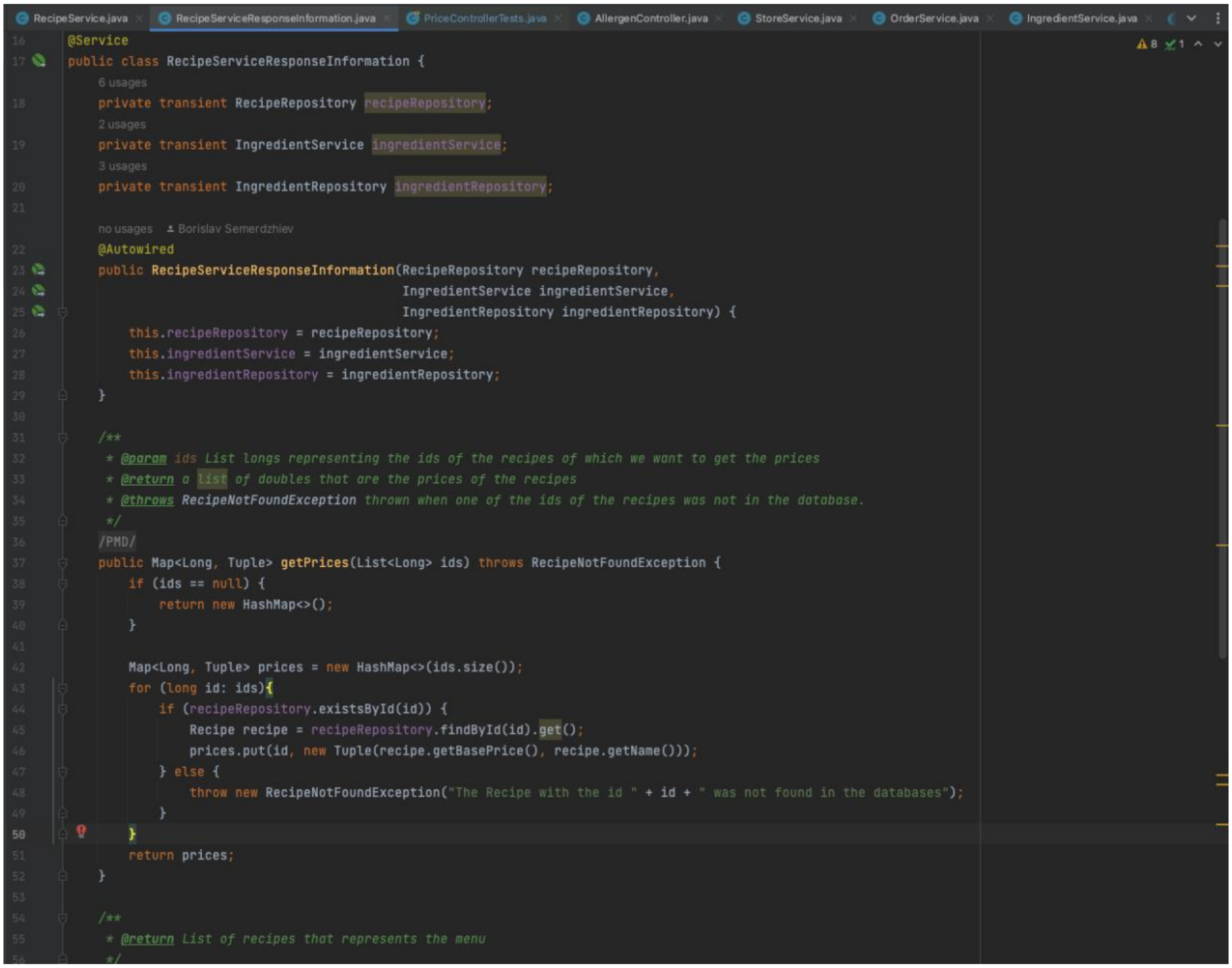


Fig 47, 48, 49: Code before refactoring

RecipeService.java | RecipeServiceResponseInformation.java | PriceControllerTests.java | AllergenController.java | StoreService.java | OrderService.java | IngredientService.java

```java
@Service
public class RecipeServiceResponseInformation {
    6 usages
    private transient RecipeRepository recipeRepository;
    2 usages
    private transient IngredientService ingredientService;
    3 usages
    private transient IngredientRepository ingredientRepository;

    no usages  Borislav Semerdzhiev
    @Autowired
    public RecipeServiceResponseInformation(RecipeRepository recipeRepository,
                                            IngredientService ingredientService,
                                            IngredientRepository ingredientRepository) {
        this.recipeRepository = recipeRepository;
        this.ingredientService = ingredientService;
        this.ingredientRepository = ingredientRepository;
    }

    /**
     * @param ids List longs representing the ids of the recipes of which we want to get the prices
     * @return a list of doubles that are the prices of the recipes
     * @throws RecipeNotFoundException thrown when one of the ids of the recipes was not in the database.
     */
    /PMD/
    public Map<Long, Tuple> getPrices(List<Long> ids) throws RecipeNotFoundException {
        if (ids == null) {
            return new HashMap<>();
        }

        Map<Long, Tuple> prices = new HashMap<>(ids.size());
        for (long id: ids){
            if (recipeRepository.existsById(id)) {
                Recipe recipe = recipeRepository.findById(id).get();
                prices.put(id, new Tuple(recipe.getBasePrice(), recipe.getName()));
            } else {
                throw new RecipeNotFoundException("The Recipe with the id " + id + " was not found in the databases");
            }
        }
        return prices;
    }

    /**
     * @return List of recipes that represents the menu
     */
```




```java
@Service
public class RecipeServiceResponseInformation {
    6 usages
    private transient RecipeRepository recipeRepository;
    2 usages
    private transient IngredientService ingredientService;
    3 usages
    private transient IngredientRepository ingredientRepository;

    no usages  Borislav Semerdzhiev
    @Autowired
    public RecipeServiceResponseInformation(RecipeRepository recipeRepository,
                                            IngredientService ingredientService,
                                            IngredientRepository ingredientRepository) {
        this.recipeRepository = recipeRepository;
        this.ingredientService = ingredientService;
        this.ingredientRepository = ingredientRepository;
    }

    /**
     * @param ids List longs representing the ids of the recipes of which we want to get the prices
     * @return a list of doubles that are the prices of the recipes
     * @throws RecipeNotFoundException thrown when one of the ids of the recipes was not in the database.
     */
    /PMD/
    public Map<Long, Tuple> getPrices(List<Long> ids) throws RecipeNotFoundException {
        if (ids == null) {
            return new HashMap<>();
        }

        Map<Long, Tuple> prices = new HashMap<>(ids.size());
        for (long id: ids){
            if (recipeRepository.existsById(id)) {
                Recipe recipe = recipeRepository.findById(id).get();
                prices.put(id, new Tuple(recipe.getBasePrice(), recipe.getName()));
            } else {
                throw new RecipeNotFoundException("The Recipe with the id " + id + " was not found in the databases");
            }
        }
        return prices;
    }

    /**
     * @return List of recipes that represents the menu
     */
```

Fig 50, 51: Code after refactoring