Final Report Team 17c
CSE2115 - Software Engineering Methods
Bookmania: User microservice

Ivar van Loon   Matei Stănescu   Maria Cristescu   Maria Tudor
Wishaal Kanhai

January 18, 2024

1 Introduction

This final report consists of some introduction and relevant information, and also the Final Requirements, Design Patterns, Software Quality, Mutation Testing and Integration Reports. Each report will analyze several relevant aspects, including rationale for certain choices we've made and detailed information. The agenda's, minutes and sprint retrospectives can be found in the GitLab wiki page of our project, these will not be handled in this report. The wiki also contains all previous said reports.

The Final Requirements report will document the list of requirements our microservice contains. The Design Patterns report will detail the design patterns we have chosen to implement, consequently explaining why, where and how we did this. The Software Quality report will analyze the quality of our project code using software metrics, this report will also show what refactorings/modifications we've made to improve the software quality. The Mutation Testing report will analyze the mutation testing scores of our code and how we have improved this, this report also shows the Jacoco coverage of our project. The Integration report will explain our experiences while integrating the other two microservices, the Bookshelf and Review microservices, with our microservice.

Each subsequent report will follow the same format, they have been merged together into one pdf file.

Final Requirements Report

Team 17c - CSE2115

January 18, 2024

1 Introduction

This report will announce all requirements, functional and non-functional, which our microservice has implemented, using the regular ordering on priorities (must, should, could), the requirements may be described as if they were not implemented (shall/must/should), but this is done just to adhere to the format of the requirements engineering assignment from week 2 of the course. We will leave out requirements implemented by other microservices, main functionalities left out include the book management of admins/authors and user reviews.

2 Terminology

1. **(Regular) User:**
    - Can add books to their bookshelf.
    - Can follow other users.
    - Can write book reviews.
    - Can manage their own account and profile
2. **Admin:**
    - Has multiple responsibilities, including adding, editing, and removing books from the overall collections.
    - Admins are users as well.
3. **Author:**
    - Can add their own books.
    - Authors are users as well.
4. **System:**
    - The User microservice.

3 Functional Requirements

3.1 Must Haves

- A client must be able to create a user account.
- A user must contain a username, a unique email address, and a password.
- A user must be able to manage their account (username, email, password)
- A user must be able to manage their profile (name, bio, profile picture, location, favorite book genres, favorite book).
- A user must be able to view other users' profile (if public).

- Admins must be able to remove, add, update, deactivate, activate and ban users.
- A user shall not be able to edit other users' information/account.
- An author must be able to prove their author identity.
- An admin must be able to prove their admin identity.

3.2 Should Haves

- A user should be able to edit privacy preferences.
- A user should be able to modify account security settings.
- A user should be able to discover other users based on name, interests, etc.
- A user should be able to reactivate their own account, if they themselves deactivated it.
- A user should be able to add genres to their favorite genre list, and also be able to remove them.
- A user should be able to set one book as their favorite, and also be able to remove it from their favorite.
- The system should implement features to deactivate or delete user accounts upon request, ensuring compliance with privacy regulations and user preferences.

3.3 Could Haves

- A user should be able to discover other users based on connections (e.g. friendships).
- A user can follow and unfollow other users, while adhering to users' privacy settings.
- The system can collect and provide analytics on the amount of (active) users, amount of authors, most popular users, most popular book and genres, and general (user) activities in a log-based way to admins.

3.4 Won't Haves

- The system will currently not have a two-factor authentication option.
- The system will currently not display notifications.
- The system will currently not improve the user experience based on analytics.

4 Non-Functional Requirements

- The system shall be modular, adding functionality should be easy.
- The system shall be easily integrated with other systems.
- The system shall be developed using Java 15.
- The system shall be developed using Spring Boot.
- The builds shall be realized using Gradle.
- A user shall not be allowed to use the same email address as another user.
- A username shall start with an alphabetic character and not include special characters (e.g., punctuation marks, $, /, -, _, etc.).

Design Patterns Report

Team 17c - CSE2115

January 18, 2024

1 Introduction

In this report, we will introduce the design patterns we have implemented. We will give a short description of what the design pattern's purpose is, then an explanation of what the pattern solves for us. We will also show a UML diagram of the implemented design patterns and links to issues/commits and classes which implemented the design patterns.

2 Design Patterns

2.1 Pattern 1: Chain of Responsibility

2.1.1 Description

This pattern allows us to pass a request along a chain of handlers, giving more than one object a chance to handle the request. This is useful since it reduces coupling. This pattern is used to handle the retrieval of user analytics, which regards analytics on several aspects of user activities. The analytics include the following information:

1. Amount of users in the system
2. Amount of active users in the system
3. Amount of authors in the system
4. A list of the top 3 followed users
5. A fraction of a global log, containing just the last 10 entries of this log as a list.
6. The most popular book in the system, based on the favorite books of users.
7. The top 3 most popular genres, based on the favorite genres of users, as a list.

Points 1, 2, and 3 require highly similar functionality; thus, they have been placed in a single handler.

2.1.2 Problem(s) solved

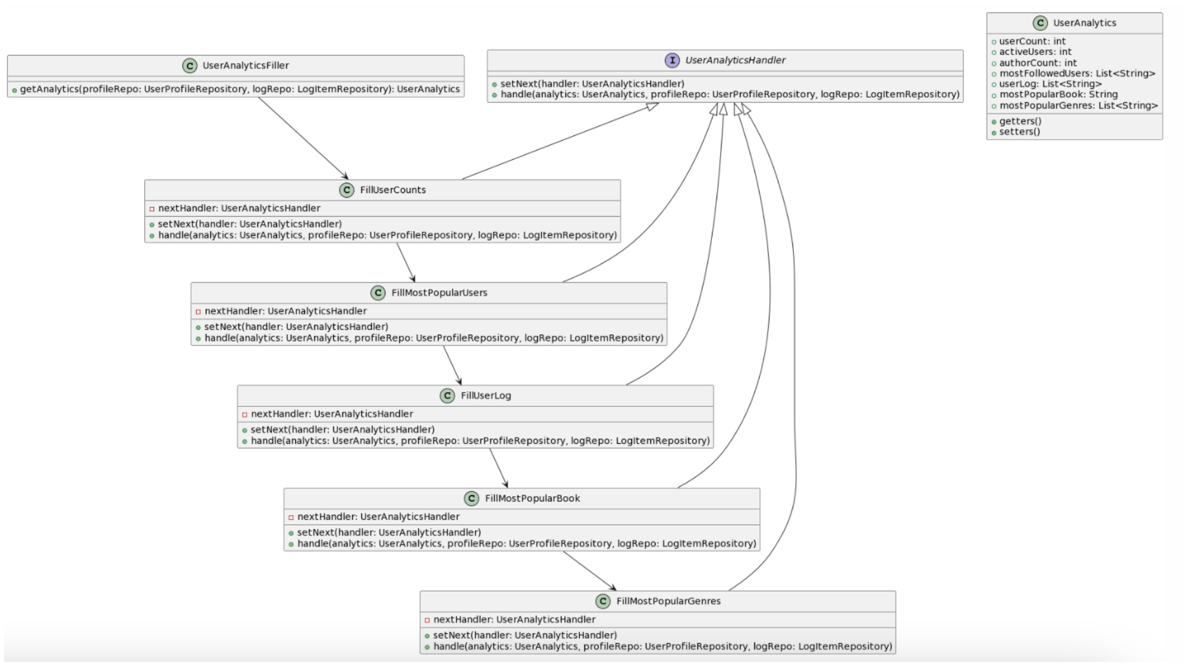
This pattern allows us to use the handlers separately, which is useful for different scenarios, maybe we only want to know one piece of information on user analytics. We did not have enough time to implement the functionality for this in our codebase. This design pattern also allows us to split the required functionality for the analytics into separate handlers instead of having one big object with a ton of responsibilities. The pattern also allows for more scalability, improved software quality, and easier integration with further development of our application.

2.1.3 UML diagram

Figure 1 contains the UML diagram of this pattern.

Our base object, namely UserAnalyticsFiller, uses a chain of handlers, each having a separate functionality and implementing an interface called UserAnalyticsHandler, to fill a new object UserAnalytics, which encompasses all information needed. The order of handlers shown in the diagram is based on the order of the attributes in UserAnalytics and are able to be interchanged.

Our chain does not have a "bail out" system, that is because the separate responsibilities have functionality to still fill the UserAnalytics in cases of errors and exceptions by filling the object with informational data where needed. For instance, if no most popular book is able to be found, the handler will set the corresponding attribute to "There is no most popular book" instead of throwing an exception. The handlers additionally require JPA repositories to be passed, one of the repositories holds the profiles of users, the other one holds the append-only log.



UserAnalyticsFiller
getAnalytics(profileRepo: UserProfileRepository, logRepo: LogItemRepository): UserAnalytics
UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)
UserAnalytics
userCount: int
activeUsers: int
authorCount: int
mostFollowedUsers: List<String>
userLog: List<String>
mostPopularBook: String
mostPopularGenres: List<String>
getters()
setters()
FillUserCounts
nextHandler: UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)
FillMostPopularUsers
nextHandler: UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)
FillUserLog
nextHandler: UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)
FillMostPopularBook
nextHandler: UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)
FillMostPopularGenres
nextHandler: UserAnalyticsHandler
setNext(handler: UserAnalyticsHandler)
handle(analytics: UserAnalytics, profileRepo: UserProfileRepository, logRepo: LogItemRepository)

Figure 1: UML diagram of our Chain of Responsibility pattern

2.1.4 Links to commits/issues and classes

- Issue 51 - Create the Chain of Responsibility design pattern
- analytics package - the package containing all classes implementing this design pattern

The issue above is a problem associated with a merge request (MR) linked to it, which actually implements this design pattern by separating the large object we initially had into separate handlers. The MR has one (1) commit in it, which contains the changes made.

2.2 Pattern 2: (Protection) Proxy

2.2.1 Description

This pattern allows us to return placeholder/proxy objects without revealing sensitive attributes of the real object. We created an UserProxy object which holds a real User object and has methods which regulate access to this real user. Our usage of the Proxy design pattern could be categorized under the Protection Proxies.

2.2.2 Problem(s) solved

Users are able to view their own User object as a UserWithoutPassword object. For security reasons, users' passwords are never retrievable through any of the endpoints.

- **User:** The real subject class that contains sensitive information such as passwords.
- **UserProxy:** The proxy class that controls access to the real User object. It holds a reference to the real User (*realUser*) and ensures that users can only view their own User object as a UserWithoutPassword object.
- **UserWithoutPassword:** A class representing the user object without the password. This is what users are allowed to view through the proxy.

2.2.3 UML diagram

Figure 2 contains the UML diagram of this pattern.

A simplified AccountController is displayed in the UML diagram. This controller has a method viewAccount() which is able to retrieve the UserWithoutPassword object using the UserProxy class.

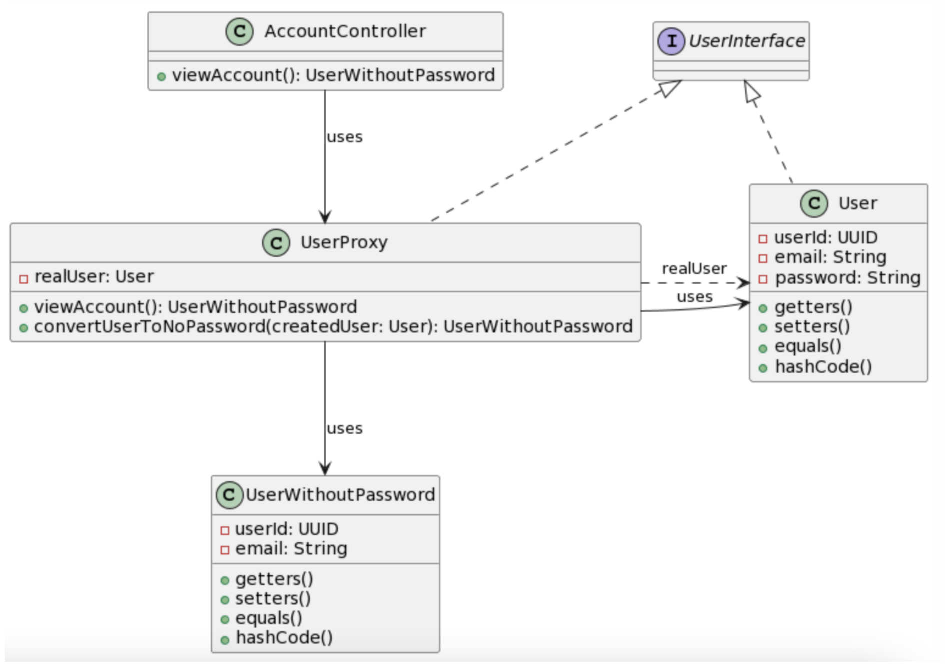


Figure 2: UML diagram of our Proxy design pattern

2.2.4 Links to commits/issues and classes

- Issue 52 - Create the Proxy design pattern The issue above is a problem associated with a merge request (MR) linked to it, which actually implements this design pattern by creating a new interface and a new class, used to protect the password information of a user.
- UserInterface.java - The UserInterface interface
- UserProxy.java - The UserProxy class

Software Quality Report

Team 17c - CSE2115

January 18, 2024

1 Introduction

In this report, we will analyze our code base using a software quality software, namely MetricsTree (version 2023.3.0), a plug-in for IntelliJ as also shown in the lectures. We will analyze our code using this software and draw conclusions based on the several metrics provided by the plug-in, we will use the CK metrics and Code Complexity. From there, we will pick out several classes and/or methods which we think are relevant for refactoring purposes. MetricsTree is able to show a lot of information on several scopes, we will only use a (small) subset of metrics, namely the metrics which are relevant to (the scope of) our project and the metrics discussed during the lectures.

We will explain what changes need to be made/have been made and which issues/commits are related to these changes. We will then show the results of the metric software again and analyze on whether we actually made improvements and whether we stumbled across issues.

Some classes or methods which need refactoring according to the metrics will be ignored, since some classes/methods are not able to be refactored at all. This can have several factors, one of them being the fact that it will change the behavior of the program, which is exactly what we don't want to happen by refactoring. In other cases, there is absolutely no way to refactor the class and still keep the functionality it holds, we encountered this problem often with some models with a lot of attributes and few methods, when adding getters and setters for these classes, the cohesion of the class will drop (i.e. the LCOM increases).

Note: MetricsTree indicates certain metrics with either a green, yellow or red circle, to indicate the severity/mildness of a metric. We mainly focused on the metrics indicated as red, since MetricsTree is quite sensitive to classifying a metric as mild/yellow and are often just false alarms. Of course, we won't neglect the yellow indicators at any point. We will also not show every change and every detail we've made on refactoring, but rather stick to some relevant/big refactorings we made.

2 Metric results: Before

See figure 1. It is notable that our project has a big amount of classes which contain long methods (20) (num greater or equal than 10). We also have a couple of classes in our project with too many methods (7) (LOC greater or equal than 16). The other groups have been deemed irrelevant or too small to be able to start a refactoring process. We will analyze said 2 groups further now.

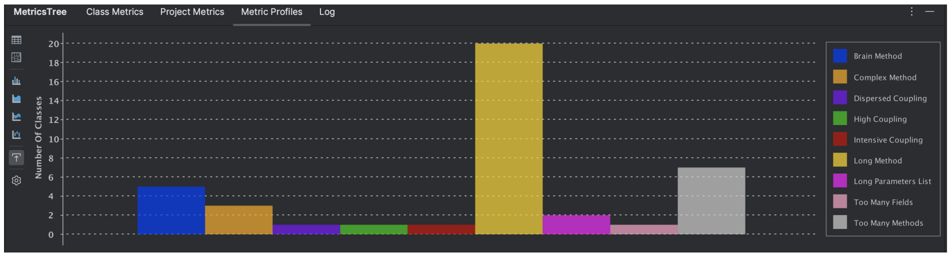
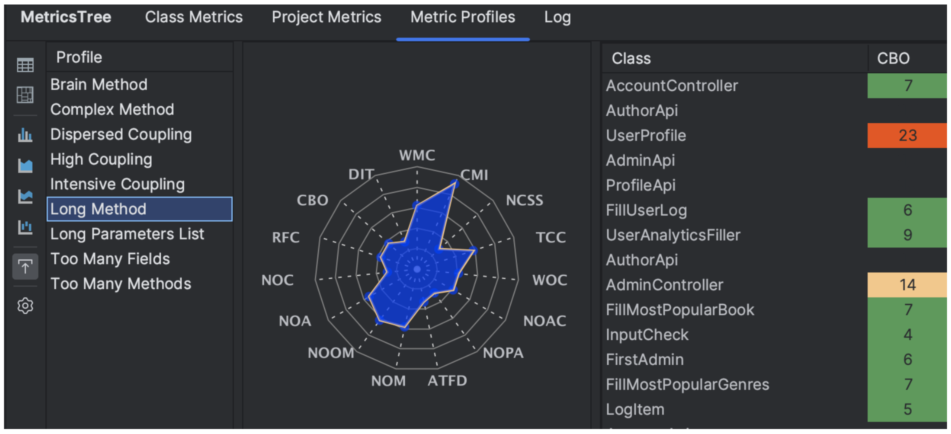


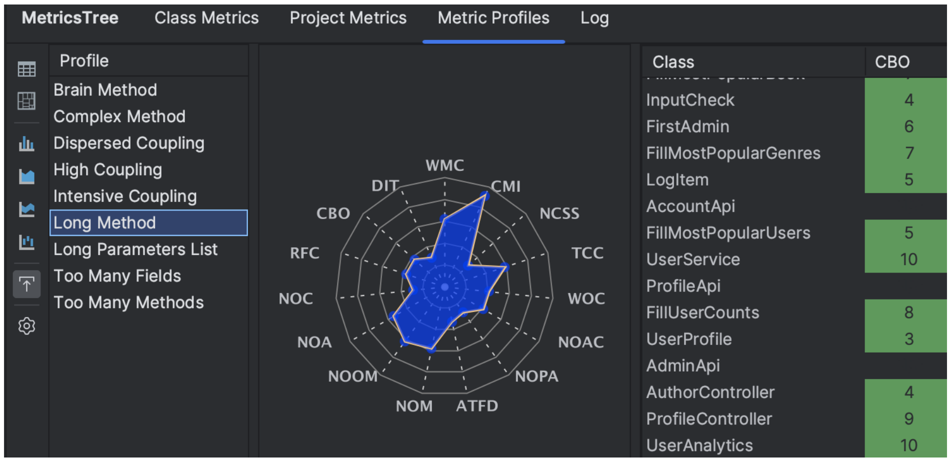
Figure 1: Metrics Profile before refactoring

See figures 2 and 3. Figures 2 and 3 show our classes with long methods (they're split in 2 figures since they didn't fit in one single screenshot). It is notable that some generated files are included in this, these are the "Api" classes and the UserProfile class which has a very high CBO score as well, which we can luckily ignore. It is also noticeable that all of our Spring controller classes contain long methods. Furthermore, AdminController has a moderate CBO score of 14 which we want to improve on.



Class	CBO
AccountController	7
AuthorApi	
UserProfile	23
AdminApi	
ProfileApi	
FillUserLog	6
UserAnalyticsFiller	9
AuthorApi	
AdminController	14
FillMostPopularBook	7
InputCheck	4
FirstAdmin	6
FillMostPopularGenres	7
LogItem	5

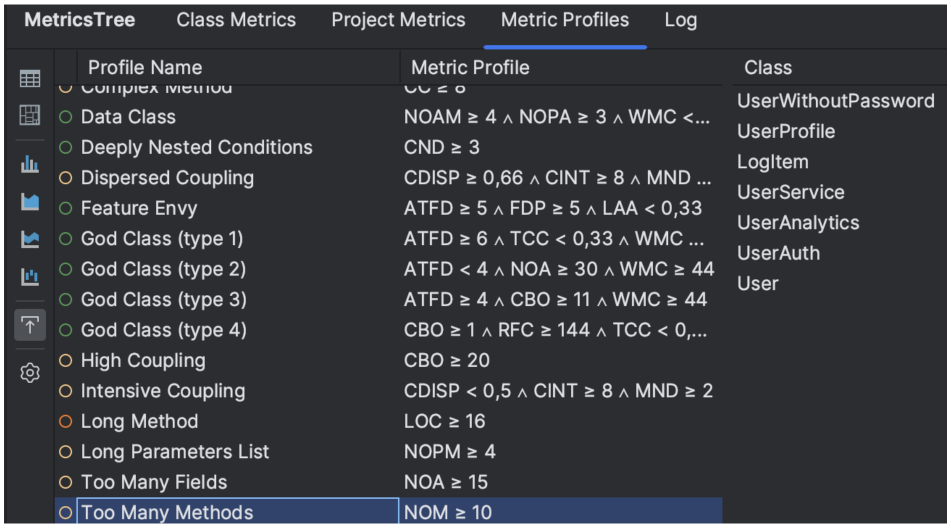
Figure 2: Classes containing long methods, alongside their CBO score (Coupling Between Objects) - First part



Class	CBO
InputCheck	4
FirstAdmin	6
FillMostPopularGenres	7
LogItem	5
AccountApi	
FillMostPopularUsers	5
UserService	10
ProfileApi	
FillUserCounts	8
UserProfile	3
AdminApi	
AuthorController	4
ProfileController	9
UserAnalytics	10

Figure 3: Classes containing long methods, alongside their CBO score (Coupling Between Objects) - Second part

Figure 4 shows all classes which contain too many methods (at least 10 methods). There are actually 4 generated files included here, those are UserWithoutPassword, UserProfile, UserAnalytics and User. That leaves us with just 3 classes with too many methods. We will try shortening these classes if possible



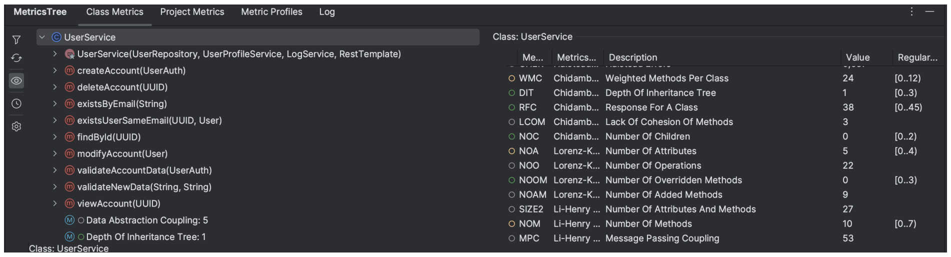
Profile Name	Metric Profile
Complex Method	CC ≥ 8
Data Class	NOAM ≥ 4 ∧ NOPA ≥ 3 ∧ WMC <...
Deeply Nested Conditions	CND ≥ 3
Dispersed Coupling	CDISP ≥ 0,66 ∧ CINT ≥ 8 ∧ MND ...
Feature Envy	ATFD ≥ 5 ∧ FDP ≥ 5 ∧ LAA < 0,33
God Class (type 1)	ATFD ≥ 6 ∧ TCC < 0,33 ∧ WMC ...
God Class (type 2)	ATFD < 4 ∧ NOA ≥ 30 ∧ WMC ≥ 44
God Class (type 3)	ATFD ≥ 4 ∧ CBO ≥ 11 ∧ WMC ≥ 44
God Class (type 4)	CBO ≥ 1 ∧ RFC ≥ 144 ∧ TCC < 0,...
High Coupling	CBO ≥ 20
Intensive Coupling	CDISP < 0,5 ∧ CINT ≥ 8 ∧ MND ≥ 2
Long Method	LOC ≥ 16
Long Parameters List	NOPM ≥ 4
Too Many Fields	NOA ≥ 15
Too Many Methods	NOM ≥ 10

Class
UserWithoutPassword
UserProfile
LogItem
UserService
UserAnalytics
UserAuth
User

Figure 4: Classes containing too many methods

2.1 Classes to be refactored

In figure 5, the metrics of the UserService class is visible. This class holds functionality for several core user actions, such as creation, deletion and modification. We can see that the weighted methods in this class exceeds the regular amount of 12, our class has 24. We can also see that the LCOM is 3, while the rest of our classes ranges between 0 and 1, and in some cases 2. We can thus conclude that 3 is an outlier, and we want to improve this. The other CK metrics seem fine for this class, those are DIT (Depth Of Inheritance Tree), RFC (Response For A Class) and NOC (Number Of Children).



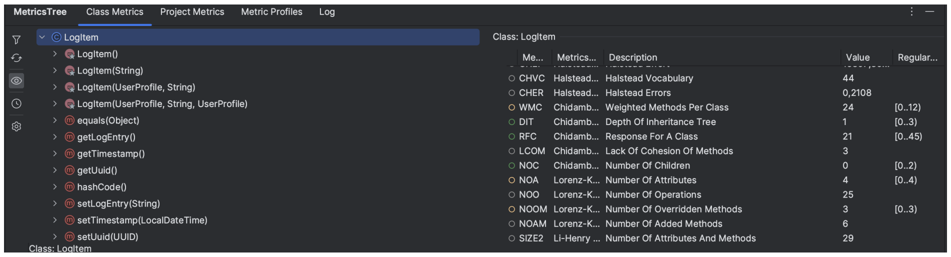
```
UserService
  UserService(UserRepository, UserProfileService, LogService, RestTemplate)
  createAccount(UserAuth)
  deleteAccount(UUID)
  existsByEmail(String)
  existsUserSameEmail(UUID, User)
  findById(UUID)
  modifyAccount(User)
  validateAccountData(UserAuth)
  validateNewData(String, String)
  viewAccount(UUID)
    Data Abstraction Coupling: 5
    Depth Of Inheritance Tree: 1
Class: UserService
```

Class: UserService

Me...	Metrics...	Description	Value	Regular...
WMC	Chidamb...	Weighted Methods Per Class	24	[0..12)
DIT	Chidamb...	Depth Of Inheritance Tree	1	[0..3)
RFC	Chidamb...	Response For A Class	38	[0..45)
LCOM	Chidamb...	Lack Of Cohesion Of Methods	3	
NOC	Chidamb...	Number Of Children	0	[0..2)
NOA	Lorenz-K...	Number Of Attributes	5	[0..4)
NOO	Lorenz-K...	Number Of Operations	22	
NOOM	Lorenz-K...	Number Of Overridden Methods	0	[0..3)
NOAM	Lorenz-K...	Number Of Added Methods	9	
SIZE2	Li-Henry ...	Number Of Attributes And Methods	27	
NOM	Li-Henry ...	Number Of Methods	10	[0..7)
MPC	Li-Henry ...	Message Passing Coupling	53	

Figure 5: UserService class with metrics

In figure 6, the metrics of the LogItem class is visible. This class holds an entity used for our log. We can see that the weighted methods in this class also exceeds the regular amount of 12, this class has 24 too. We can also see that the LCOM is also 3, and we want to improve this. The other CK metrics seem fine for this class.



```
LogItem
  LogItem()
  LogItem(String)
  LogItem(UserProfile, String)
  LogItem(UserProfile, String, UserProfile)
  equals(Object)
  getLogEntry()
  getTimestamp()
  getUuid()
  hashCode()
  setLogEntry(String)
  setTimestamp(LocalDateTime)
  setUuid(UUID)
Class: LogItem
```

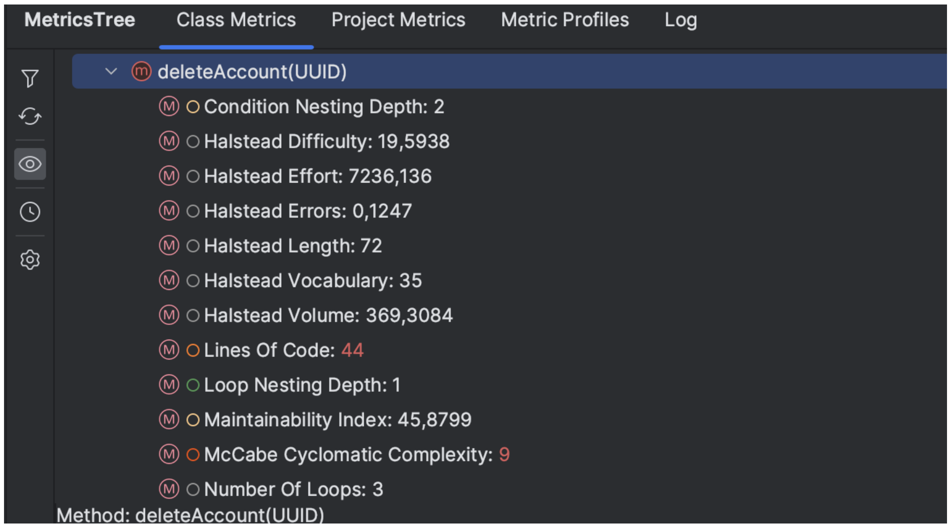
Class: LogItem

Me...	Metrics...	Description	Value	Regular...
CHVC	Halstead...	Halstead Vocabulary	44	
CHER	Halstead...	Halstead Errors	0,2108	
WMC	Chidamb...	Weighted Methods Per Class	24	[0..12)
DIT	Chidamb...	Depth Of Inheritance Tree	1	[0..3)
RFC	Chidamb...	Response For A Class	21	[0..45)
LCOM	Chidamb...	Lack Of Cohesion Of Methods	3	
NOC	Chidamb...	Number Of Children	0	[0..2)
NOA	Lorenz-K...	Number Of Attributes	4	[0..4)
NOO	Lorenz-K...	Number Of Operations	25	
NOOM	Lorenz-K...	Number Of Overridden Methods	3	[0..3)
NOAM	Lorenz-K...	Number Of Added Methods	6	
SIZE2	Li-Henry ...	Number Of Attributes And Methods	29	

Figure 6: LogItem class with metrics

2.2 Methods to be refactored

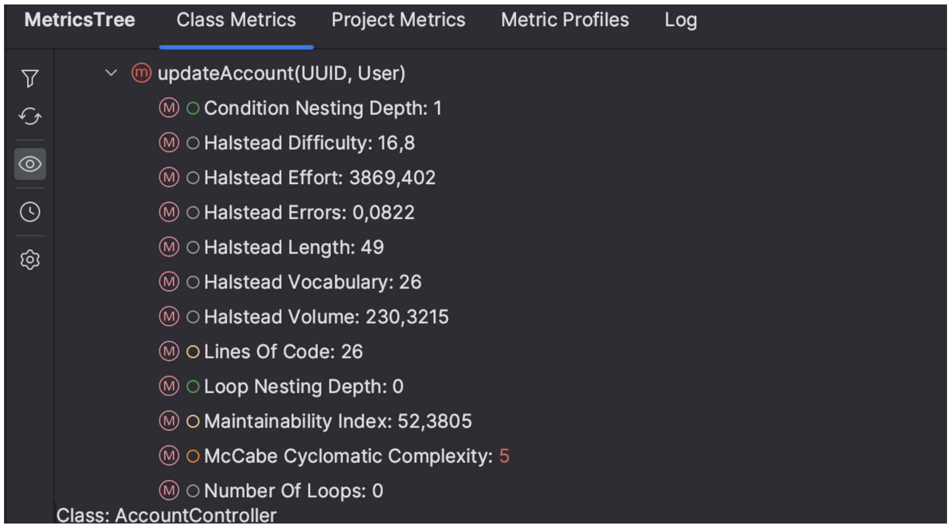
In figure 7, the metrics of the deleteAccount method (which resides in the UserService) are visible. This method deletes users' accounts. The amounts are already printed in red, signalling a bad sign, but the lines of code is pretty large and the cyclomatic complexity is 9, which is also very large for CC. We want to reduce both of these variables.



```
deleteAccount(UUID)
    Condition Nesting Depth: 2
    Halstead Difficulty: 19,5938
    Halstead Effort: 7236,136
    Halstead Errors: 0,1247
    Halstead Length: 72
    Halstead Vocabulary: 35
    Halstead Volume: 369,3084
    Lines Of Code: 44
    Loop Nesting Depth: 1
    Maintainability Index: 45,8799
    McCabe Cyclomatic Complexity: 9
    Number Of Loops: 3
Method: deleteAccount(UUID)
```

Figure 7: deleteAccount function with metrics - UserService

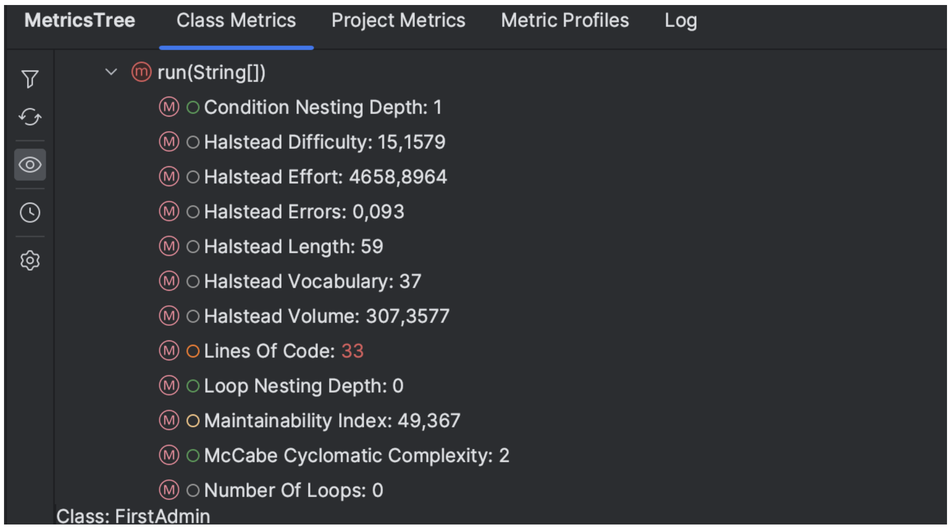
In figure 8, the metrics of the updateAccount method (which also resides in the UserService) are visible. This method updates users' accounts. It is noticeable that the cyclomatic complexity for this method is too large as well. We want to reduce this.



```
updateAccount(UUID, User)
    Condition Nesting Depth: 1
    Halstead Difficulty: 16,8
    Halstead Effort: 3869,402
    Halstead Errors: 0,0822
    Halstead Length: 49
    Halstead Vocabulary: 26
    Halstead Volume: 230,3215
    Lines Of Code: 26
    Loop Nesting Depth: 0
    Maintainability Index: 52,3805
    McCabe Cyclomatic Complexity: 5
    Number Of Loops: 0
Class: AccountController
```

Figure 8: updateAccount function with metrics - UserService

Lastly, in figure 9, the metrics of the "run" method (which resides in the FirstAdmin component) is visible. This method creates an admin on startup. We can see that the lines of code exceeds the wanted amount of maximum 30. This is confirmed by us, since we actually created this component in a rush and copy pasted some functionality from one of our controllers. We will be able to reduce the lines of codes in this method.



```
run(String[])
    Condition Nesting Depth: 1
    Halstead Difficulty: 15,1579
    Halstead Effort: 4658,8964
    Halstead Errors: 0,093
    Halstead Length: 59
    Halstead Vocabulary: 37
    Halstead Volume: 307,3577
    Lines Of Code: 33
    Loop Nesting Depth: 0
    Maintainability Index: 49,367
    McCabe Cyclomatic Complexity: 2
    Number Of Loops: 0
Class: FirstAdmin
```

Figure 9: run function with metrics - FirstAdmin

3 Metric results: During and After

During the refactoring process we found that some classes were wrongly classified as having long methods, this is mainly due to the fact that MetricsTree uses LOC instead of eLOC. We formatted our code to be readable, and thus have occasionally left some white-spaces to indicate code which are (highly) related to each other. As a consequence, the LOC increases, while the eLOC would've stayed constant. MetricsTree does not seem to have functionality for eLOC.

We also put some code on new lines for readability, we mainly did this for stream operations, such as mappings. We are able to decrease the lines of code very easily, but it would just not make sense/is contradictory to do it from a software quality perspective, so we left most of our classes with a high LOC alone.

Here are the results for the classes/methods we did try to refactor:

3.1 Links to commits/issues

The following linked issues do not contain all the refactoring we made, we actually refactored our code piece by piece during the project and did not assign specific issues to them.

- issue 67 - ”Refactor deleteAccount() method in UserService”
- related commit - ”Moved some code to new methods”
- issue 61 - ”Refactor run() method in FirstAdmin component”
- related commit - ”Refactored FirstAdmin”
- issue 59 - ”Refactor Author Controller and similar users functionality + tests”
- related commit - ”Refactored into services and tested all the endpoints that I have been assigned”
- issue 56 - ”Refactor AccountController”
- related commit - ”Refactor code into Services”
- commit - ”Move the analytics filler to own classes”

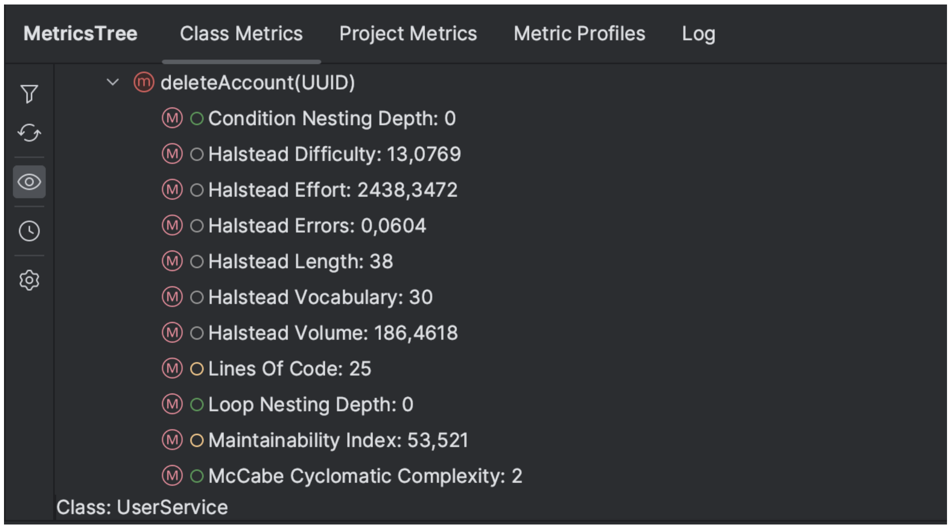
3.2 Classes refactored

We did not manage to refactor the LogItem class, this is due to the fact that this class forms an entity for our log. The LogItem contains metadata, such as the actual log entry and timestamp. The class has just 3 methods, i.e. 3 constructors for ease of use. Because of the many getters and setters this class has by comaprison to the actual methods in the class, the LCOM will increase are not able to improve this without altering the functionalities.

We have also not been able to refactor the UserService class, this is mainly due to the fact that this service holds many seperate responsibilities regarding not only User's, but also their UserProfile's. All of these responsiblities/methods are needed to make our program function. As a consequence of the many independent methods, the LCOM is 3 which we are not able to modify without changing the entirety of the applications' functionality, so we left it as is. The WMC is therefore also still above the treshold of 12, it's 26.

3.3 Methods refactored

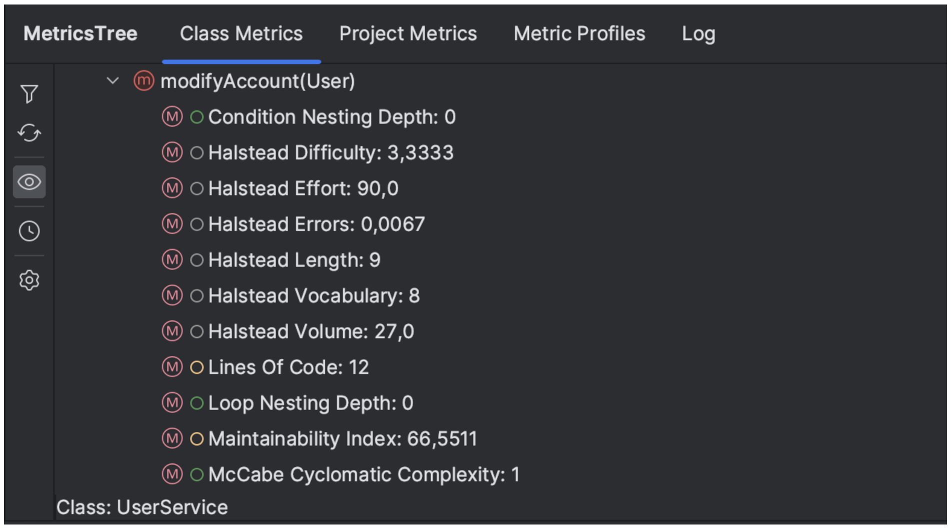
In figure 10, the metrics of the deleteAccount method are seen again. We managed to improve the LOC and the cyclomatic complexity. We managed to do this by removing some redundant whitespaces and moving some business logic to separate methods which held one responsibility each. We moved the code, since the deleteAccount() method was too complex by all the checks on users' following lists etcetera.



```
deleteAccount(UUID)
    Condition Nesting Depth: 0
    Halstead Difficulty: 13,0769
    Halstead Effort: 2438,3472
    Halstead Errors: 0,0604
    Halstead Length: 38
    Halstead Vocabulary: 30
    Halstead Volume: 186,4618
    Lines Of Code: 25
    Loop Nesting Depth: 0
    Maintainability Index: 53,521
    McCabe Cyclomatic Complexity: 2
Class: UserService
```

Figure 10: deleteAccount function with metrics - UserService

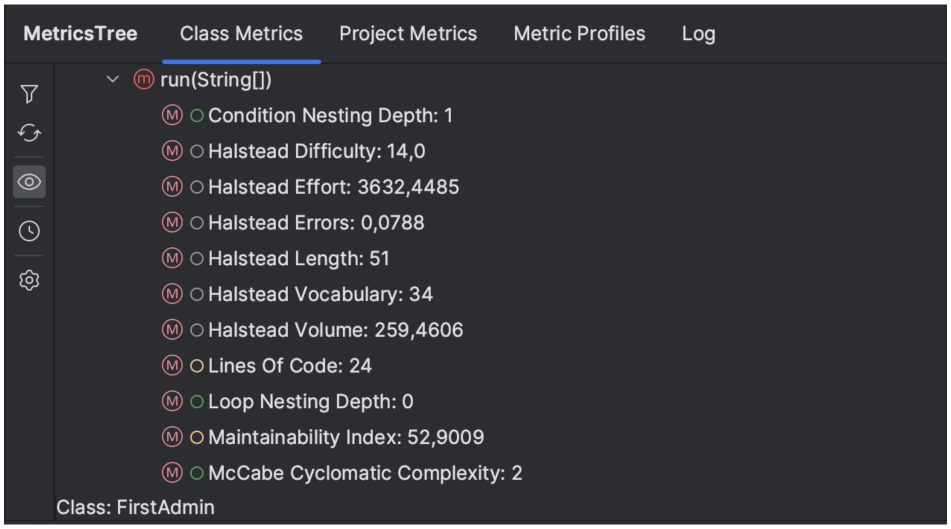
In figure 11, the metrics of the modifyAccount method are seen. We actually updated the name from 'updateAccount' to 'modifyAccount' since it seemed to be a more appropriate name. We managed to decrease the cyclomatic complexity from 5 down to 1. Additionally our LOC decreased from 26 all the way down to 12. We did this by removing some duplicated business logic and instead make calls to existing methods.



```
modifyAccount(User)
    Condition Nesting Depth: 0
    Halstead Difficulty: 3,3333
    Halstead Effort: 90,0
    Halstead Errors: 0,0067
    Halstead Length: 9
    Halstead Vocabulary: 8
    Halstead Volume: 27,0
    Lines Of Code: 12
    Loop Nesting Depth: 0
    Maintainability Index: 66,5511
    McCabe Cyclomatic Complexity: 1
Class: UserService
```

Figure 11: updateAccount function with metrics - UserService

Lastly, in figure 12, the metrics of the run() method are seen again. As said before, this method contained tons of duplicated code and by removing this duplicated code and making method calls to existing functions, we managed to decrease the LOC from 33 to 24. We did this by making use of our UserService which has functionality for creating a new account, exactly what we needed for creating our first admin on start-up.



```
run(String[])
    Condition Nesting Depth: 1
    Halstead Difficulty: 14,0
    Halstead Effort: 3632,4485
    Halstead Errors: 0,0788
    Halstead Length: 51
    Halstead Vocabulary: 34
    Halstead Volume: 259,4606
    Lines Of Code: 24
    Loop Nesting Depth: 0
    Maintainability Index: 52,9009
    McCabe Cyclomatic Complexity: 2
Class: FirstAdmin
```

Figure 12: run function with metrics - FirstAdmin

3.3.1 Project Metrics

See figure 13. The colors of the metrics changed, but the content has stayed same. It is noticeable that the group of classes with long methods is still large, even larger (24) than what we started with (20). The classes with too many methods has also increased from 7 to 8. This is not immediately a bad sign, since we actually added a lot more functionalities to our application since we started refactoring. This caused us to have more complex methods, longer methods and more methods overall.

We have scanned the metrics of each class and every method and have noticed that no method actually has a huge LOC (no methods with LOC greater or equal than 30), besides some generated classes. As said before, MetricsTree is very quick to classify a method as being too long, but after manual checks we are actually very happy with the results we've achieved.

The classes with too many methods mainly consisted of entities, most of which were generated, and one controller and the UserService class. It is logical that the entities have a lot of methods, caused by all the getters, setters and equals methods. Furthermore, we are actually not able to reduce the methods in the UserService. As said earlier, the UserService holds a lot of responsibilities which we are not able to distribute anywhere without changing the program's functionality, which results in this class having 11 methods, above the threshold of 10.

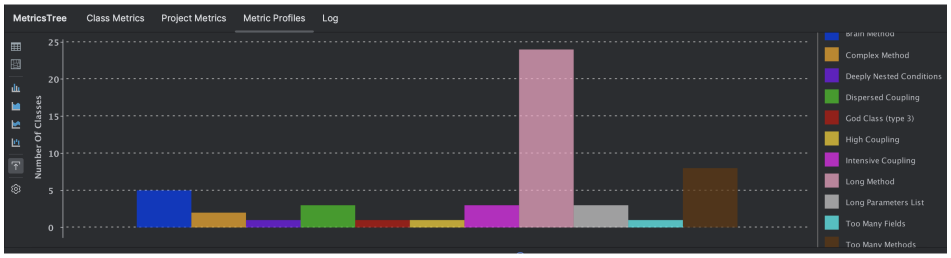
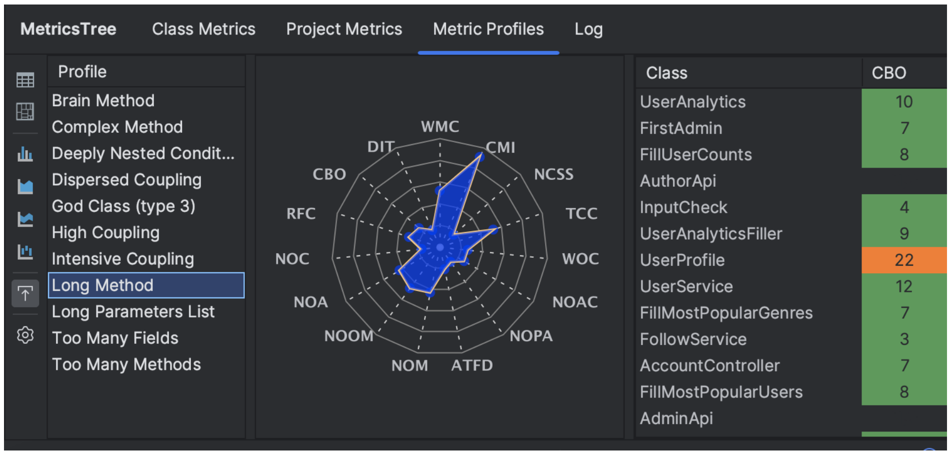


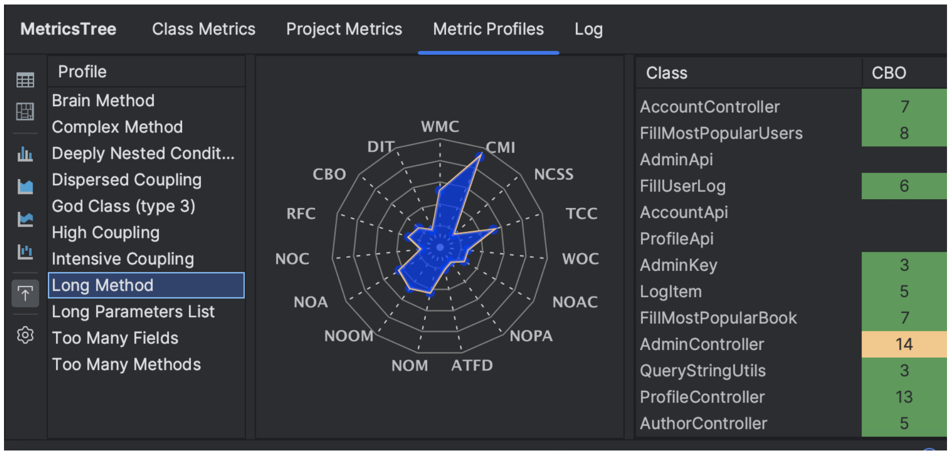
Figure 13: Metrics Profile after refactoring

See figures 14 and 15. The classes with long methods are seen again alongside their CBO scores. The CBO scores did not seem to have been impacted by our refactoring, not for the worse nor better. We did try lowering the CBO score for the AdminController, but no luck. The AdminController is the controller which holds most 'power' over the application, it's logical since admins are able to do almost anything in the program. As a result, the controller uses many object types, such as User's, UserProfile's, LogItem's, AdminKey's, and so on, resulting in a moderately high CBO score.



Class	CBO
UserAnalytics	10
FirstAdmin	7
FillUserCounts	8
AuthorApi	
InputCheck	4
UserAnalyticsFiller	9
UserProfile	22
UserService	12
FillMostPopularGenres	7
FollowService	3
AccountController	7
FillMostPopularUsers	8
AdminApi	

Figure 14: Classes containing long methods, alongside their CBO score (Coupling Between Objects) - First part



Profile
Brain Method
Complex Method
Deeply Nested Condit...
Dispersed Coupling
God Class (type 3)
High Coupling
Intensive Coupling
Long Method
Long Parameters List
Too Many Fields
Too Many Methods

Class	CBO
AccountController	7
FillMostPopularUsers	8
AdminApi	
FillUserLog	6
AccountApi	
ProfileApi	
AdminKey	3
LogItem	5
FillMostPopularBook	7
AdminController	14
QueryStringUtils	3
ProfileController	13
AuthorController	5

Figure 15: Classes containing long methods, alongside their CBO score (Coupling Between Objects) - Second part

Mutation Testing Report

Team 17c - CSE2115

January 18, 2024

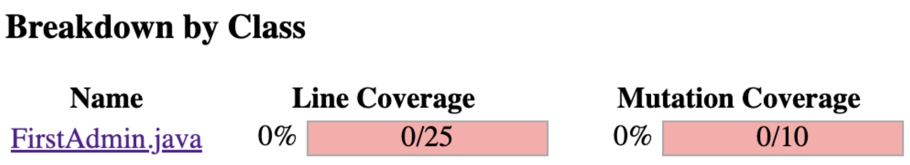
1 Introduction

In this report we will analyze our codebase based on results from mutation testing software, more specifically PITest (version 1.5.1). Firstly, we will show what these test results were before we started on any refactoring/further testing. After we have done that, we will analyze said results and make conclusions on how to improve. Lastly, we will explain what changes we have made to the code/test code and why this worked/didn't work and we will show what the results of the mutation testing software were after the refactoring. During this report we will provide several links to issues and commits which regards the refactoring and further testing of our code to improve the mutation testing results. We will exclude any classes/code that we did not change, since this is what is also specified in the lab manual.

The package names in the figures are not truly correct, we renamed the packages to say 'user' instead of 'example'

2 Test results: Before

2.1 Components

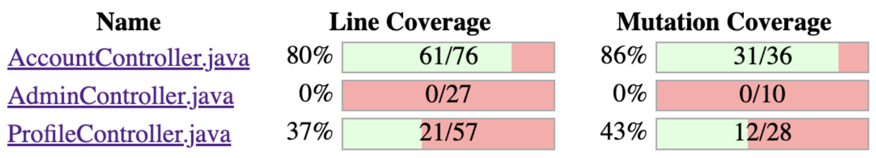


Breakdown by Class

Name	Line Coverage		Mutation Coverage	
FirstAdmin.java	0%	0/25	0%	0/10

Figure 1: Mutation score of the classes in the components package before modifications

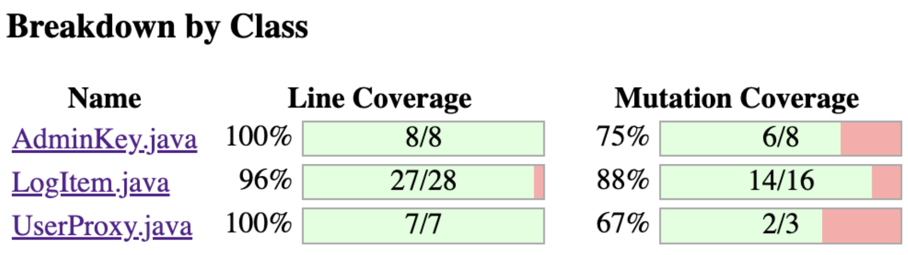
2.2 Controllers



Name	Line Coverage		Mutation Coverage	
AccountController.java	80%	61/76	86%	31/36
AdminController.java	0%	0/27	0%	0/10
ProfileController.java	37%	21/57	43%	12/28

Figure 2: Mutation score of the classes in the controllers package before modifications

2.3 Models

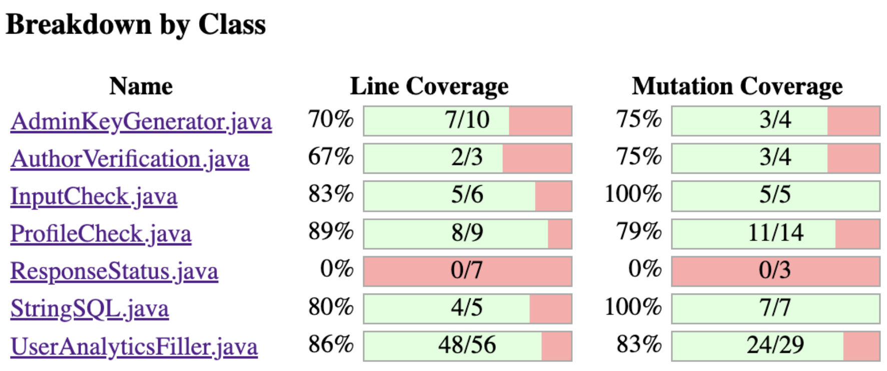


Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AdminKey.java	100%	8/8	75%	6/8
LogItem.java	96%	27/28	88%	14/16
UserProxy.java	100%	7/7	67%	2/3

Figure 3: Mutation score of the classes in the models package before modifications

2.4 Services



Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AdminKeyGenerator.java	70%	7/10	75%	3/4
AuthorVerification.java	67%	2/3	75%	3/4
InputCheck.java	83%	5/6	100%	5/5
ProfileCheck.java	89%	8/9	79%	11/14
ResponseStatus.java	0%	0/7	0%	0/3
StringSQL.java	80%	4/5	100%	7/7
UserAnalyticsFiller.java	86%	48/56	83%	24/29

Figure 4: Mutation score of the classes in the services package before modifications

3 Analysis: Before

It is notable throughout the figures that the java classes we did test (with a line coverage above 0) have a reasonable mutation test score, so we can conclude that the tests we did create are able to reach (almost) all mutations in the tested code. Of course, that is not enough, we will have to test our code more thoroughly and make sure we have a reasonable line coverage across all classes.

So, throughout several issues/commits, we tried optimizing this mutation score and consequently the line coverage as well. A couple of issues were reserved just to solve this mutation score, other several commits were pushed throughout the project to increase the mutation score by default. Of course, the figures shown before do not show all classes which are present in the final result, because we developed the project further and added new classes as a result, but most are shown.

3.1 Related Issues and Commits

3.1.1 Related Issues

- issue 53 - "Test the Admin and Profile controllers"
- issue 54 - "Test the UserAnalyticsFiller 'pipeline'"
- issue 55 - "Improve testing AccountController"
- issue 64 - "Create integration tests for AdminController"

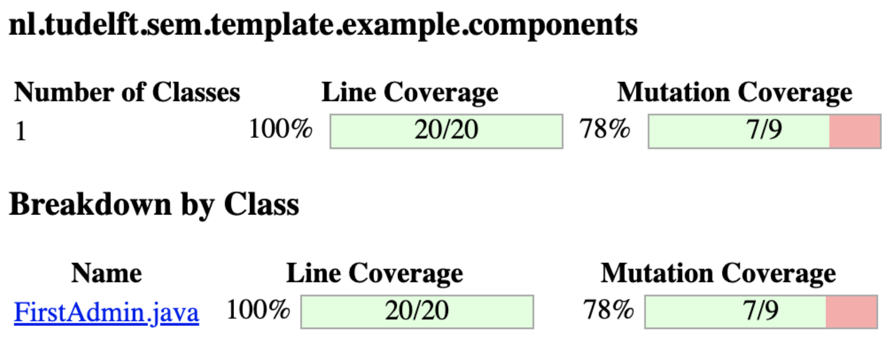
3.1.2 Related Commits

- commit 1 - ”Improve mutation testing percentage”
- commit 2 - ”Used PITest to increase mutation score”
- commit 3 - ”Improve coverage”
- commit 4 - ”Improved mutation score”

4 Test results: After

4.1 Components

We created new tests that are able to test the Spring Component ’FirstAdmin’ which creates an admin at startup. We weren’t able to kill all mutants since there is a call to add a shutdown hook to the runtime, which we cannot reach in a test.



nl.tudelft.sem.template.example.components

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	20/20	78%	7/9

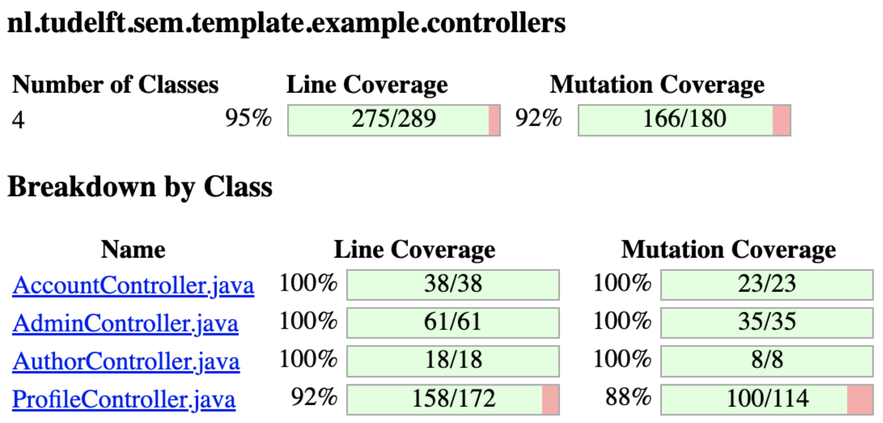
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
FirstAdmin.java	100%	20/20	78%	7/9

Figure 5: Mutation score of the classes in the components package after modifications

4.2 Controllers

We created new tests which were able to test most methods in the ProfileController and roughly half of the methods in AdminController, this was done in issue 53. The mutation scores have improved a lot by comparison and we will continue to improve this. There are just a few mutations left to kill. PITest also had 3 time-outs (thus, 3 mutants which might not have been killed) during the tests of the AdminController, but the mutation score would still be very good incase these mutants would still be alive.



nl.tudelft.sem.template.example.controllers

Number of Classes	Line Coverage		Mutation Coverage	
4	95%	275/289	92%	166/180

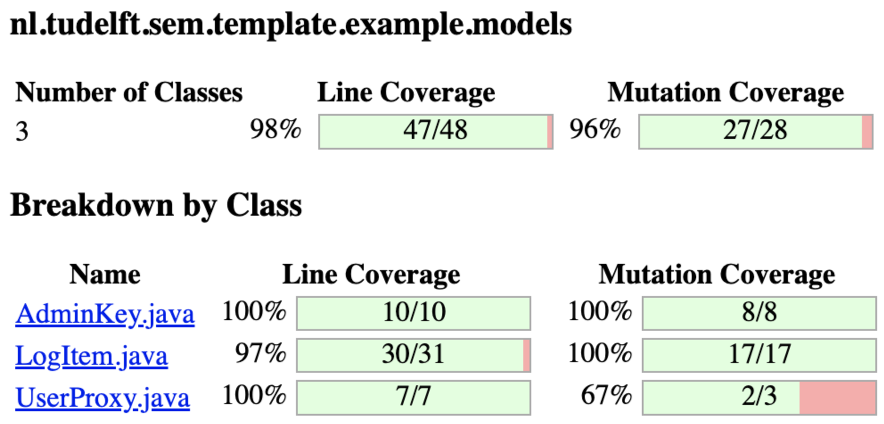
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AccountController.java	100%	38/38	100%	23/23
AdminController.java	100%	61/61	100%	35/35
AuthorController.java	100%	18/18	100%	8/8
ProfileController.java	92%	158/172	88%	100/114

Figure 6: Mutation score of the classes in the controllers package after modifications

4.3 Models

Not much changed to the models package, we did manage to kill a couple of alive mutants, but that's all. This package was already pretty well tested to begin with.



nl.tudelft.sem.template.example.models

Number of Classes	Line Coverage		Mutation Coverage	
3	98%	47/48	96%	27/28

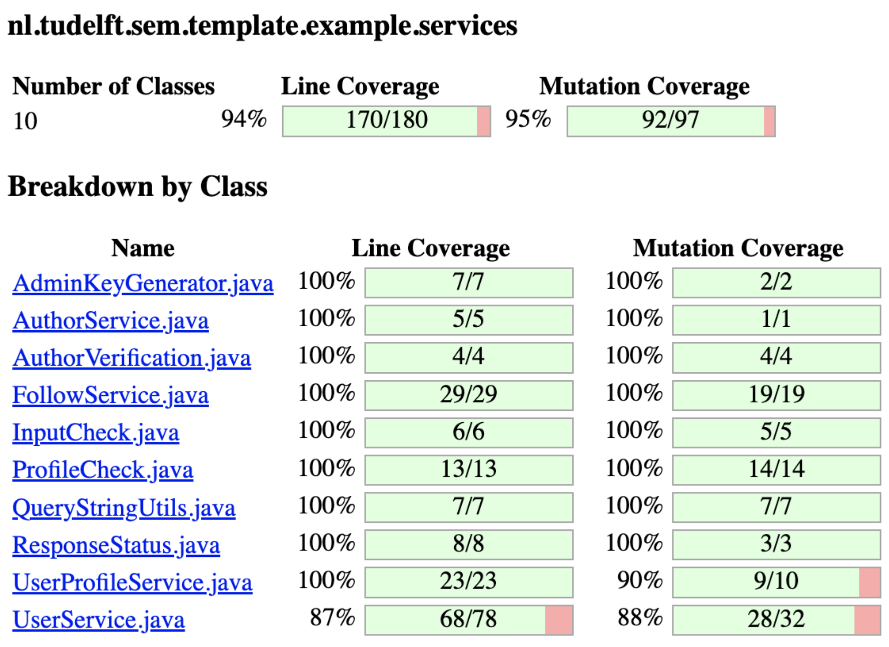
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AdminKey.java	100%	10/10	100%	8/8
LogItem.java	97%	30/31	100%	17/17
UserProxy.java	100%	7/7	67%	2/3

Figure 7: Mutation score of the classes in the models package after modifications

4.4 Services

We managed to kill almost all mutants in this package, achieving a mutation score of 95 percent. The line coverage is not optimal according to these results, but we found out that PITest actually has a different line coverage result than the line coverage we got from Jacoco in some cases, so the line coverage seen in this report is not the grand truth. We also modified the UserAnalyticsFiller, this class got split up into seperate handlers forming a Chain of Responsibility design pattern. The package they have been placed in have been tested quite well according to the test results (figure 9), we managed to kill all mutants.



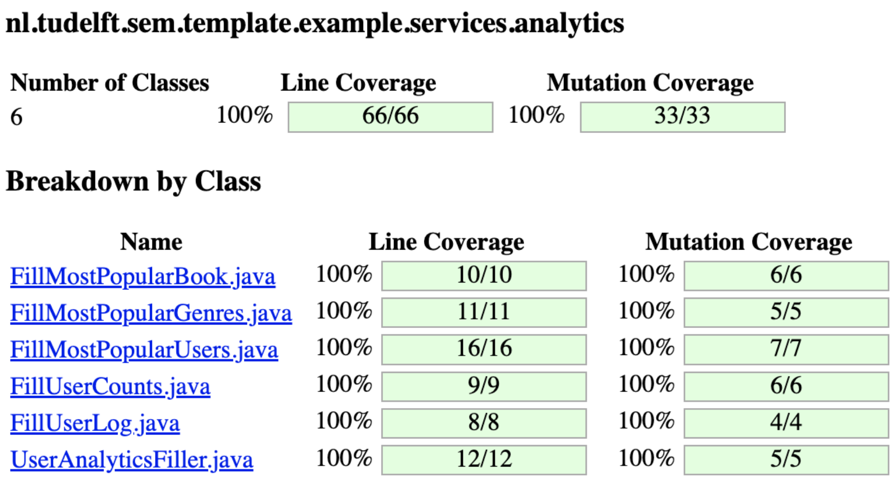
nl.tudelft.sem.template.example.services

Number of Classes	Line Coverage		Mutation Coverage	
10	94%	170/180	95%	92/97

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AdminKeyGenerator.java	100%	7/7	100%	2/2
AuthorService.java	100%	5/5	100%	1/1
AuthorVerification.java	100%	4/4	100%	4/4
FollowService.java	100%	29/29	100%	19/19
InputCheck.java	100%	6/6	100%	5/5
ProfileCheck.java	100%	13/13	100%	14/14
QueryStringUtils.java	100%	7/7	100%	7/7
ResponseStatus.java	100%	8/8	100%	3/3
UserProfileService.java	100%	23/23	90%	9/10
UserService.java	87%	68/78	88%	28/32

Figure 8: Mutation score of the classes in the services package after modifications



nl.tudelft.sem.template.example.services.analytics

Number of Classes	Line Coverage		Mutation Coverage	
6	100%	66/66	100%	33/33

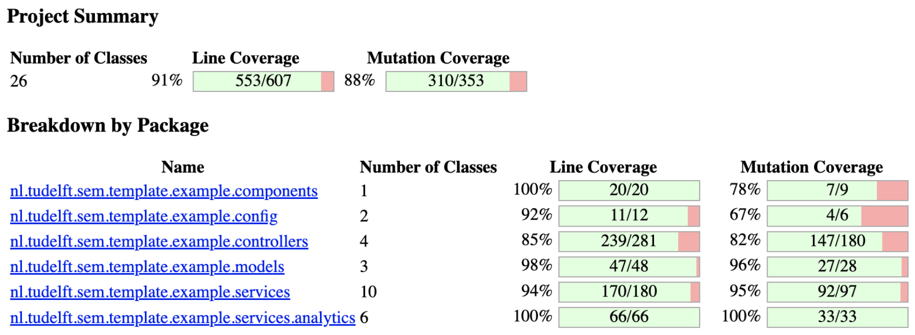
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
FillMostPopularBook.java	100%	10/10	100%	6/6
FillMostPopularGenres.java	100%	11/11	100%	5/5
FillMostPopularUsers.java	100%	16/16	100%	7/7
FillUserCounts.java	100%	9/9	100%	6/6
FillUserLog.java	100%	8/8	100%	4/4
UserAnalyticsFiller.java	100%	12/12	100%	5/5

Figure 9: Mutation score of the classes in the services.analytics package after modifications

5 Overall scores

See figure 10. We managed to get a mutation score of 88 percent (on the changed code), which is much better than the treshold of 70 percent specified in the lab manual.



Project Summary

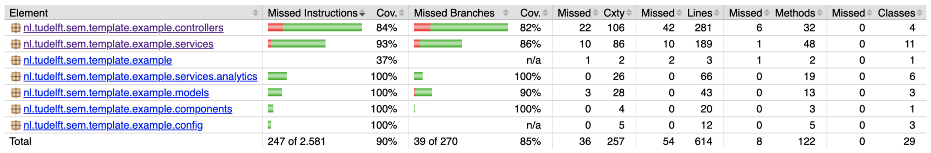
Number of Classes	Line Coverage		Mutation Coverage	
26	91%	553/607	88%	310/353

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.example.components	1	100%	20/20	78%	7/9
nl.tudelft.sem.template.example.config	2	92%	11/12	67%	4/6
nl.tudelft.sem.template.example.controllers	4	85%	239/281	82%	147/180
nl.tudelft.sem.template.example.models	3	98%	47/48	96%	27/28
nl.tudelft.sem.template.example.services	10	94%	170/180	95%	92/97
nl.tudelft.sem.template.example.services.analytics	6	100%	66/66	100%	33/33

Figure 10: Mutation score of the entire project as a whole after modifications

The Jacoco report, figure 11, shows that our project has a branch coverage of 85 percent, which is above the treshold of 80 percent specified in the lab manual. We also got a line coverage of 93 percent. These numbers may be out of date (worse than the final numbers) since we keep trying to improve the test scores.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.sem.template.example.controllers		84%		82%	22	106	42	281	6	32	0	4
nl.tudelft.sem.template.example.services		93%		86%	10	86	10	189	1	48	0	11
nl.tudelft.sem.template.example		37%		n/a	1	2	2	3	1	2	0	1
nl.tudelft.sem.template.example.services.analytics		100%		100%	0	26	0	66	0	19	0	6
nl.tudelft.sem.template.example.models		100%		90%	3	28	0	43	0	13	0	3
nl.tudelft.sem.template.example.components		100%		100%	0	4	0	20	0	3	0	1
nl.tudelft.sem.template.example.config		100%		n/a	0	5	0	12	0	5	0	3
Total	247 of 2.581	90%	39 of 270	85%	36	257	54	614	8	122	0	29

Figure 11: Jacoco scores of the entire project

Integration Report

Team 17c - CSE2115

January 18, 2024

1 Introduction

In week 7/8 of the course, we have tried integrating the other 2 microservices from the other teams, the Bookshelf (team 17a) and Reviews (team 17b) microservices, with our microservice, the User microservice. In this document we will report and reflect on how this went and what issues we may have come across and whether the integration actually succeeded.

2 Integration: Bookshelf microservice

Our microservice required some functionalities from the bookshelf microservice. Specifically, we needed their Book, Genre and Bookshelf entity functionalities.

2.1 (Un)favoring a Book

Our microservice allows users to favor a book, and also unfavoring a book. Our microservice thus needed a way to check whether a bookID (UUID) given by a user to favorite is able to be linked to an existing book. In the same way, we had to check this when a user wanted to unfavor a book using a UUID. We checked this using the RestTemplate object provided by Spring. We would send a request to the endpoint of the Bookshelf microservice to see if a UUID for a book is valid or not.

2.1.1 Paths used

- `/bookshelves/book/{bookId}` - Get a book by its ID. We used this path to see if we got a successful response using a given UUID.

2.2 (Un)favoring a Genre

Our microservice also allows users to favor a genre, and also unfavoring a book. Our microservice thus additionally needed a way to check whether a genreID (UUID) given by a user to favorite is able to be linked to an existing genre. For this functionality we also used the RestTemplate to call the corresponding endpoint of the bookshelf microservice to handle this.

2.2.1 Paths used

- `/bookshelves/genre/all` - Get all genres. We used the response of this to check whether a given UUID is contained in it.

2.3 Deleting bookshelves of a user

Our microservice allows users to delete their accounts, when doing this all bookshelves of this user have to be deleted as well. For this we use a path provided by the bookshelf microservice to delete all bookshelves of a user.

2.3.1 Paths used

- `/bookshelves/bookshelf/{userId}/removeAll/requestBy/{requesterId}` - Remove all of a user's bookshelves. We used this endpoint when deleting a user, we do not check for any responses since it's not our responsibility to check whether the removal went through.

2.4 Tests performed

2.4.1 Test 1: Add genre to Favorites

To test adding a genre to the favorites, we added a sample genre to the Bookshelf microservice with the name 'test', then with the UUID of that genre, we call our own endpoint at: `/user/profile/-favoriteGenre/{genreId}`. This is where we found a few problems, problem one: the bookshelf microservice did not prefix their URL's with /bookshelves, thus we had to adjust the URL's in our code. Problem two, however, is a bit harder to solve. Our endpoint calls the `/genre/all` endpoint of the Bookshelf microservice to test whether the given genreId exists in their database, and if it does, we add the genreId to the favoriteGenres attribute of our UserProfile. However, when trying to call their `/genre/all` endpoint, we try parsing it as an array of Strings, which results in an error since it is actually an array of Genre objects in JSON format. There was no time left to add the Genre Entity and refactor the endpoint on our end, thus we have to consider this endpoint broken integration-wise.

2.4.2 Test 2: Set book as favorite book

To test this functionality, we added a book named testbook to the Bookshelf microservice database, then we call our own endpoint at: `/profile/favoriteBook/{bookId}`. With the change we made to the URL regarding the previous test, this functionality works flawlessly. We check whether the book exists using their endpoint `GET /book/{bookId}`, then if that returns `200 OK`, we add it to the `favoriteBook` field of our User Profile. We can conclude that integration regarding this endpoint is a success.

2.4.3 Test 3: Deleting a user, needs to delete their bookshelves

To test this functionality, we created a Bookshelf entity on the Bookshelf database, and set the owner to our test user. Then we called the `/account/delete` endpoint on our end to verify whether it actually deletes the bookshelf too. In order to delete the users' bookshelves when we delete a user, we call the endpoint `/bookshelf/userId/removeAll/requestBy/requesterId` on the Bookshelf microservice. This works like a charm and didn't need any new fixes.

2.4.4 Conclusion

2/3 tests passed after slight modifications to our code. This is in our opinion a decent score since we had expected more serious problems like the one in Test 1.

3 Integration: Review microservice

Our microservice did not require any functionality from this microservice, so there was no need to integrate this microservice with ours in any way. Our microservice does not have any functionalities related to user reviews, although their microservice may need to use some functionalities from us.