SOEN 6431 - SOFTWARE COMPREHENSION AND MAINTENANCE

DELIVERABLE 2

DÉJÀ VU

Supervisor
Prof. Pankaj Kamthan

Team C
Authors
Saeed Jamalifashi- 40093292
Darshakkumar Kachchhi - 40206619
Siddhartha Kattoju - 29209905
Navdeep Kaur - 40237921

GitHub : `https://github.com/darshak-k/SOEN6431-C-SCM`

Contents

1 Introduction

Software re-engineering is a crucial process of modifying, updating, or restructuring an existing software system to improve its overall performance, maintainability, reliability, or other non-functional aspects without changing its intended functionality. For our project, we have selected the Library Management System as the candidate R (more details can be found in Deliverable-1).

The Java-based library management system serves as an effective and structured platform for automating and managing various library operations. It offers librarians a streamlined approach to handle tasks such as cataloging, circulation, managing patrons, inventory control, and generating reports. Additionally, the system enables easy searching and retrieval of books, tracking borrowing and return transactions, and ensures seamless communication between library staff and users. The ultimate objective is to improve the efficiency, accuracy, and accessibility of library services.

To identify issues in the code, automatic code identification can be utilized which involves using specialized tools such as SonarQube, algorithms, and softwares to automatically detect, recognize, and extract code segments from a given source code. Advantages of automatic identification is speed, scalability, and consistency. But, they may not be as accurate as an expert who review the code manually. Hence, manual code identification can be utilized which involves human intervention and relies on manual inspection and analysis of the source code. Developers review the code, identify code segments, and extract relevant information based on their expertise. Disadvantages of manual identification is that they are seldom time consuming and expensive.

Following a comprehensive analysis of the code base, we have detected a total of **18 significant code smells** that are adversely affecting the system's performance, maintainability, and scalability. To rectify these issues, we will apply appropriate refactoring techniques. Our approach involves dividing lengthy methods into smaller, more manageable units, removing duplicated code by extracting reusable functions or classes, simplifying complex conditionals, incorporating design patterns to enhance the system's architecture, and enhancing naming conventions for improved code readability. This proposal outlines our strategy to address these code smells and security concerns, ultimately enhancing the overall quality of the system.

2 Undesirables

Our team utilized Sonar Cloud to identify and address various issues in the software for this assignment. The findings encompassed a range of undesirables, including severe security vulnerabilities and seemingly trivial issues like commented-out blocks of code. The software enabled us to detect and quantify these problems, allowing us to effectively correct them all in our efforts to improve the overall quality of the system.

In this section, we have provided comprehensive details about the undesirables identified in the software using Team Scale. These details include the type of undesirable, category, and code smell type, all of which shed light on the reasons behind their undesirability. Additionally, each code smell is accompanied by a summary, providing further insight into the nature of the issue. In total, we discovered 18 different types of findings, encompassing a wide range of issues. To enhance understanding, we have also incorporated graphs that visually represent the distribution of these undesirables, facilitating a better grasp of their impact on the system.

We have categorized the undesirables into three types: Bugs, Vulnerabilities, and Code Smells, each with its respective explanation:

- **Bug**: A bug refers to an error, flaw, or unintended behavior in the software code that causes incorrect or unexpected results when the program is executed.

- **Vulnerability**: A vulnerability represents a weakness in the software system that could be exploited by an attacker to compromise its security, integrity, or availability.

- **Code Smell**: Code smell indicates poorly designed or written code that can make it challenging to maintain, understand, or extend. While it may not directly lead to bugs, it negatively impacts code quality and increases the likelihood of introducing defects.

1	Examination	Usage of ’Stream.collect(Collectors.toList())’ with ’Stream.toList()
	Summary	’Stream.collect(Collectors.toList())’ instead of the more concise and modern ’Stream.toList()’, leading to reduced readability, maintainability, and missed opportunities for performance improvements.
	Category	Obsolete Code
	Type	Code Smell
	Code Smell Type	Obsolete Code
	Occurrence	4

2	Examination	Constant instead of duplicating this literal "redirect:/authors" 3 times.
	Summary	User has used string literal 3 times for "redirect:/authors" and it
	Category	Discouraged APIs
	Type	Code Smell
	Code Smell Type	Duplicated Code
	Occurrence	3

3	Examination	Persistent entities are used as arguments of "@RequestMapping" methods
	Summary	Improper use of a persistent entity as an argument of a "@RequestMapping" method.
	Category	"Architecture" or "Design" issues
	Type	Vulnerability
	Code Smell Type	Inappropriate Use of Entities
	Occurrence	3

4	Examination	Logger should be specified with BookServiceImpl.java
	Summary	This practice enables a clear and communicative logger configuration. Using different conventions for naming loggers can lead to confusion in the configuration process, and using the same class name for multiple class loggers hinders the ability to configure each class's logger separately and precisely.
	Category	Logging
	Type	Code Anomalies
	Code Smell Type	Implementation smells - Attribute name and type are opposite
	Occurrence	1

5	Examination	A class should have only one responsibility or job.
	Summary	The aim is to improve code organization, maintainability, and reusability by adhering to the Single Responsibility Principle. The Application.java class should focus on starting the application and all other initialization tasks should be handled by a separate class.
	Category	Code Organization and Maintainability
	Type	Refactoring and Design Improvement
	Code Smell Type	Monolithic Class and Code Duplication
	Occurrence	1

6	Examination	Data Separation using JSON File.
	Summary	Moving data to a separate JSON file decouples it from the Java code. This allows for independent modifications of the data without requiring changes to the code. If new data needs to be added or existing data needs to be modified, it can be done by editing the JSON file, rather than modifying the Java code. Therefore, it provides better version control, ease of update.
	Category	Code Organization and Maintainability
	Type	Refactoring and Design Improvement
	Code Smell Type	Tight Coupling and Configuration in Code.
	Occurrence	1

7	Examination	At least one assertion to this test case.
	Summary	Assertions are essential in testing to validate expected outcomes and ensure that the software behaves as intended. Without assertions, the test case may not effectively verify the functionality or catch potential issues in the code.
	Category	Testing
	Type	Test Case Improvement
	Code Smell Type	Missing Assertions.
	Occurrence	1

8	Examination	Use static access with "com.opencsv.ICSVWriter"
	Summary	Utilizing static access for the constant allows direct access without creating an instance of the class, improving code readability and reducing unnecessary object creation.In Java, constants are typically defined as static and final variables to represent fixed values that should not change during the execution of the program.
	Category	Coding Best Practice
	Type	Code Optimization
	Code Smell Type	Static Access with Constants.
	Occurrence	2

9	Examination	Constant instead of duplicating this literal "category" 5 times.
	Summary	User has used string literal 5 times for model attribute. Utilizing constants to represent commonly used string values enhances code maintainability, readability, and reduces the risk of typographical errors.
	Category	Discouraged APIs
	Type	Code Smell
	Code Smell Type	Duplicated Code
	Occurrence	5

10	Examination	Immediately return this expression instead of assigning it to the temporary variable "categoryVo".
	Summary	The recommended practice is to directly return the expression itself, avoiding the overhead of creating and maintaining temporary variables.The variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned.
	Category	Coding Best Practice
	Type	Code Optimization
	Code Smell Type	Unnecessary Temporary Variables
	Occurrence	1

11	Examination	Define a constant instead of duplicating this literal "redirect:/categories" 3 times.
	Summary	The recommended practice is to directly return the expression itself, avoiding the overhead of creating and maintaining temporary variables.The variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned.
	Category	Coding Best Practice
	Type	Code Smell
	Code Smell Type	Unnecessary Temporary Variables
	Occurrence	3

12	Examination	Replace persistent entity with a simple POJO or DTO object.
	Summary	One of the primary reasons for this recommendation is to achieve a clear separation between different layers of the application.
	Category	Coding Best Practice
	Type	Code Smell
	Code Smell Type	Data clumps
	Occurrence	3

13	Examination	Define a constant instead of duplicating this literal "redirect:/publishers".
	Summary	Instead of using the literal string "redirect:/publishers" multiple times in the code, it is recommended to create a constant variable with a meaningful name to represent the same value.
	Category	Coding Best Practice
	Type	Code Smell
	Code Smell Type	Duplicated Code
	Occurrence	3

14	Examination	Replace "publisher" which is a persistent entity with a simple POJO or DTO object.
	Summary	The code is using an entity class for the Publisher object and it recommends to use a POJO or a DTO object instead of an entity.
	Category	Coding Best Practice
	Type	Code Smell
	Code Smell Type	Data clumps
	Occurrence	1

15	Examination	Replace the type specification in the constructor call with the diamond operator ("<>").
	Summary	It is recommended to use the diamond operator (<>) for type inference when creating an instance of the PageImpl class.
	Category	Coding Best Practice
	Type	Code Smell
	Code Smell Type	Unnecessary Generic
	Occurrence	1

16	Examination	Add a private constructor to hide the implicit public one.
	Description	Utility classes should not have public constructors Sonar Rules(Java:S1118)
	Type	Code Smell
	Occurrence	1

17	Examination	Call "item.isPresent()" or "!item.isEmpty()" before accessing the value.
	Description	Optional value should only be accessed after calling isPresent() Sonar Rules(Java:S3655)
	Type	Code Smell
	Occurrence	2

18	Examination	Call "item.isPresent()" or "!item.isEmpty()" before accessing the value.
	Summary	Optional value should only be accessed after calling isPresent() Sonar Rules(Java:S3655)
	Type	Code Smell
	Occurrence	2

19	Examination	Inject the user service directly into "authenticationProvider", the only method that uses it
	Description	Factory method injection should be used in "@Configuration" classes java:S3305
	Type	Code Smell
	Occurrence	1

20	Examination	Define a constant instead of duplicating the literal "redirect:/books" 3 times.
	Description	String literals should not be duplicated java:S1192
	Type	Code Smell
	Occurrence	1

21	Examination	Replace the author persistent entity with a simple POJO or DTO object
	Description	Persistent entities should not be used as arguments of "@RequestMapping" methods java:S4684
	Type	Vulnerability
	Occurrence	Once per entity

3 Re-engineering Methods for Undesirables

Re-engineering, refers to the process of analyzing and modifying an existing software system to improve its structure, functionality, or other non-functional attributes. The goal of re-engineering is to enhance the software's maintainability, understandability, performance, or other aspects without changing its general function.

In this section, we have presented our discoveries and linked them to their corresponding reengineering rules, rule types, tags, undesirable severity, undesirable likelihood, and the team member responsible for refactoring. To gather this information, we performed an analysis of the forum sonar source, where we matched the findings to the most relevant reengineering rules.

Our system was entirely built in Java, Spring Boot, and we utilized Java static code analysis to identify unique rules related to bugs, vulnerabilities, security hotspots, and code smells in our Java code. This helped us correctly assess the value of each finding and prioritize our refactoring efforts accordingly.

No.	Field	Value
1	Findings	Usage of 'Stream.collect(Collectors.toList())' with 'Stream.toList()
	Description	"Stream.toList()" method should be used instead of "collectors" when an unmodifiable list is needed
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Major
	Likelihood	Medium
	Source	Sonar Rules(RSPEC-6204)
	Technical Debt	5 min effort
	Refactored By	Darshak Kachchhi

No.	Field	Value
2	Findings	Constant instead of duplicating this literal "redirect:/authors" 3 times
	Description	String literals should not be duplicated. The code should use a named constant instead of duplicating the literal "redirect:/categories" three times, as it enhances maintainability and readability by centralizing the value and allows for easy modifications in the future if needed
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Critical
	Likelihood	High
	Source	Sonar Rules(RSPEC-1192)
	Technical Debt	8 min effort
	Refactored By	Darshak Kachchhi

No.	Field	Value
3	Findings	Persistent entities are used as arguments of ”@RequestMapping” methods
	Description	Improper use of a persistent entity as an argument of a ”@RequestMapping” method. To mitigate it, the persistent entity should be replaced with a simple POJO or DTO object to avoid potential security and data integrity risks
	Re-engineering Rule Type	Vulnerability
	Undesirable Severity	Critical
	Likelihood	Rare
	Source	Sonar Rules(RSPEC-4684)
	Technical Debt	10 min effort
	Refactored By	Darshak Kachchhi

No.	Field	Value
4	Findings	Logger should be specified with BookServiceImpl.java
	Description	This practice enables a clear and communicative logger configuration. Using different conventions for naming loggers can lead to confusion in the configuration process, and using the same class name for multiple class loggers hinders the ability to configure each class’s logger separately and precisely.
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Minor
	Likelihood	Rare
	Source	Sonar Rules(RSPEC-1192)
	Technical Debt	5 min effort
	Refactored By	Darshak Kachchhi

No.	Field	Value
5	Findings	A class should have only one responsibility or job.
	Description	The aim is to improve code organization, maintainability, and re-usability by adhering to the Single Responsibility Principle. The Application.java class should focus on starting the application and all other initialization tasks should be handled by a separate class.
	Re-engineering Rule Type	Refactoring and Design Improvement
	Undesirable Severity	Medium
	Likelihood	Rare
	Source	Manual Identification
	Technical Debt	25 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
6	Findings	Data Separation using JSON File.
	Description	Moving data to a separate JSON file decouples it from the Java code. This allows for independent modifications of the data without requiring changes to the code. If new data needs to be added or existing data needs to be modified, it can be done by editing the JSON file, rather than modifying the Java code. Therefore, it provides better version control, ease of update.
	Re-engineering Rule Type	Refactoring and Design Improvement
	Undesirable Severity	High
	Likelihood	Rare
	Source	Manual Identification
	Technical Debt	30 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
7	Findings	At least one assertion to this test case.
	Description	A test case without assertions ensures only that no exceptions are thrown. Beyond basic runnability, it ensures nothing about the behavior of the code under test.
	Re-engineering Rule Type	Test Case Improvement
	Undesirable Severity	low
	Likelihood	Rare
	Source	Tests should include assertions java:S2699
	Technical Debt	10 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
8	Findings	Use static access with "com.opencsv.ICSVWriter".
	Description	In the interest of code clarity, static members of a base class should never be accessed using a derived type's name. Doing so is confusing and could create the illusion that two different static members exist.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	medium
	Likelihood	Common
	Source	"static" base class members should not be accessed via derived types java:S3252
	Technical Debt	5 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
9	Findings	Constant instead of duplicating this literal "category" 5 times
	Description	String literals should not be duplicated. The code should use a named constant instead of duplicating the literal ”redirect:/categories” three times, as it enhances maintainability and readability by centralizing the value and allows for easy modifications in the future if needed
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Critical
	Likelihood	High
	Source	Sonar Rules(RSPEC-1192)
	Technical Debt	12 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
10	Findings	Immediately return this expression instead of assigning it to the temporary variable "categoryVo".
	Description	Declaring a variable only to immediately return or throw it is a bad practice.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	Medium
	Likelihood	High
	Source	Sonar Rules(Java:S1488)
	Technical Debt	2 min effort
	Refactored By	Navdeep Kaur

No.	Field	Value
12	Findings	Replace persistent entity with a simple POJO or DTO object.
	Description	It is recommended to use POJOs or DTOs in the presentation layer of your Spring application to ensure that the interaction with simple POJOs or DTOs rather than the persistent entities.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	Critical
	Likelihood	High
	Source	Sonar Rules(Java:S1488)
	Technical Debt	10 min effort
	Refactored By	Saeed Jamalifashi

No.	Field	Value
13	Findings	Define a constant instead of duplicating this literal "redirect:/publishers".
	Description	Using a constant instead of duplicating the literal string helps improve readability and maintainability, code reusability, and consistency.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	Critical
	Likelihood	High
	Source	Sonar Rules(Java:S1192)
	Technical Debt	10 min effort
	Refactored By	Saeed Jamalifashi

No.	Field	Value
14	Findings	Replace this persistent entity with a simple POJO or DTO object.
	Description	To maintain a clear separation between the layers and simplify the data representation used in different parts of the application it is recommended to use a DTO or POJO object.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	Critical
	Likelihood	High
	Source	Sonar Rules(Java:S4684)
	Technical Debt	10 min effort
	Refactored By	Saeed Jamalifashi

No.	Field	Value
15	Findings	Replace the type specification in the constructor call with the diamond operator ("<>").
	Description	It is recommended to use diamond operator to avoid explicitly specifying the type arguments for generic classes when the compiler can infer the types from the context.
	Re-engineering Rule Type	Code Improvement
	Undesirable Severity	Minor
	Likelihood	low
	Source	Sonar Rules(Java:S2293)
	Technical Debt	1 min effort
	Refactored By	Saeed Jamalifashi

No.	Field	Value
16	Findings	Add a private constructor to hide the implicit public one.
	Description	Utility classes should not have public constructors Sonar Rules(Java:S1118)
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Major :Utility classes, which are collections of static members, are not meant to be instantiated. Even abstract utility classes, which can be extended, should not have public constructors. Java adds an implicit public constructor to every class which does not define at least one explicitly. Hence, at least one non-public constructor should be defined.
	Status	Closed (fixed) Sonar Dashboard Link
	Source	Pull Request
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

No.	Field	Value
17	Findings	Call "item.isPresent()" or "!item.isEmpty()" before accessing the value.
	Description	Optional value should only be accessed after calling isPresent() Sonar Rules(Java:S3655)
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Major :Optional value can hold either a value or not. The value held in the Optional can be accessed using the get() method, but it will throw a NoSuchElementException if there is no value present. To avoid the exception, calling the isPresent() or ! isEmpty() method should always be done before any call to get().
	Status	Closed (fixed) Sonar Dashboard Link
	Source	Pull Request
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

No.	Field	Value
18	Findings	Call "item.isPresent()" or "!item.isEmpty()" before accessing the value.
	Description	Optional value should only be accessed after calling isPresent() Sonar Rules(Java:S3655)
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Major :Optional value can hold either a value or not. The value held in the Optional can be accessed using the get() method, but it will throw a NoSuchElementException if there is no value present. To avoid the exception, calling the isPresent() or ! isEmpty() method should always be done before any call to get().
	Status	Closed (fixed) Sonar Dashboard Link
	Source	Pull Request
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

No.	Field	Value
19	Findings	Inject the user service directly into "authenticationProvider", the only method that uses it
	Description	Factory method injection should be used in "@Configuration" classes java:S3305
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Critical :When @Autowired is used, dependencies need to be resolved when the class is instantiated, which may cause early initialization of beans or lead the context to look in places it shouldn't to find the bean. To avoid this tricky issue and optimize the way the context loads, dependencies should be requested as late as possible. Sonar Rules(Java:S3305)
	Status	Closed (fixed) Sonar Dashboard Link
	Source	Pull Request
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

No.	Field	Value
20	Findings	Define a constant instead of duplicating the literal "redirect:/books" 3 times.
	Description	String literals should not be duplicated java:S1192
	Re-engineering Rule Type	Code Smell
	Undesirable Severity	Critical : Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences. On the other hand, constants can be referenced from many places, but only need to be updated in a single place.
	Status	Closed (fixed) Sonar Dashboard Link
	Source	Sonar Rules(Java:S1192)
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

No.	Field	Value
21	Findings	Replace the author persistent entity with a simple POJO or DTO object
	Description	Persistent entities should not be used as arguments of "@RequestMapping" methods java:S4684
	Re-engineering Rule Type	Vulnerability
	Undesirable Severity	Critical : Persistent objects (@Entity or @Document) are linked to the underlying database and it's possible to feed some unexpected fields on the arguments of the @RequestMapping annotated methods potentially violating the integrity of the system
	Status	Closed (fixed)
	Source	Sonar Rules(Java:S4684)
	Technical Debt	10 min effort
	Refactored By	Sid Kattoju

4 Location of Undesirables

The table below provides a summary of the locations of all the identified undesirables in our codebase. Each undesirable is specified by mentioning the folder name, the file where it is found within that folder, and a description of where the undesirable is located, along with the number of lines of source code it encompasses.

Upon analysis, we discovered a total of 190 different findings, accounting for 2,142 lines of code that contained undesirables. Through diligent maintenance and corrective measures, we successfully reduced the number of undesirables to just 8. However, we chose to retain these 8 undesirables because altering them would have impacted the system's functionality, which was beyond the scope of the project description.

No.	File	Lines of code	Bug	Vulnerability	Code Smell
1	lms\controller \BookController.java	94			
2	src\addProduct.java	112	94	0	8
3	src\AdminPanel.java	220	189	0	15
5	lms\Application.java	9			
6	lms\ApplicationInitializer.java	52			
7	test\lms \ApplicationTests.java	15			
8	lms\service \impl\FileServiceImpl.java	66			
9	lms\controller \CategoryController.java	61			
10	lms\util\Mapper.java	35			
		LOC : 2076	**SLOC : 1721**	**31**	**1**

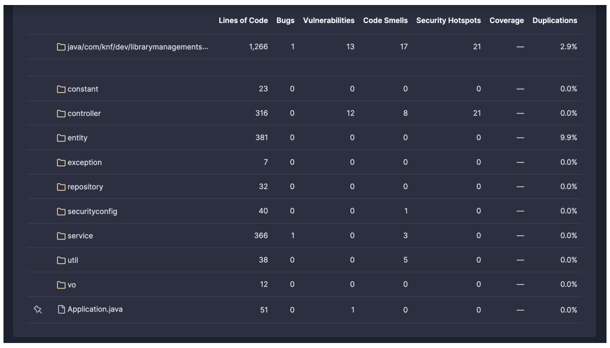
⋆⋆**Lines:** Physical lines in the file. ⋆**Lines of code:** Actual logical code

5 Software Metric log

Software metrics are measurable and quantifiable characteristics of a software system. They play a crucial role in assessing various aspects of the software, such as performance, productivity, and work planning.

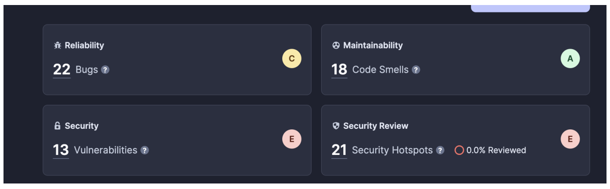
In the case of candidate R, the library management system, Sonar Cloud, a software metric analyzing tool was employed to identify differences in the system's quality before and after refactoring. The following presentation highlights the significant changes in the software metrics of candidate R resulting from the refactoring process.

Before Refactoring



	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
java/com/knf/dev/librarymanagements...	1,266	1	13	17	21	—	2.9%
constant	23	0	0	0	0	—	0.0%
controller	316	0	12	8	21	—	0.0%
entity	381	0	0	0	0	—	9.9%
exception	7	0	0	0	0	—	0.0%
repository	32	0	0	0	0	—	0.0%
securityconfig	40	0	0	1	0	—	0.0%
service	366	1	0	3	0	—	0.0%
util	38	0	0	5	0	—	0.0%
vo	12	0	0	0	0	—	0.0%
Application.java	51	0	1	0	0	—	0.0%

Figure 5.1: Package Metrics of Candidate R - Library Management System before Refactoring



Reliability
22 Bugs
C

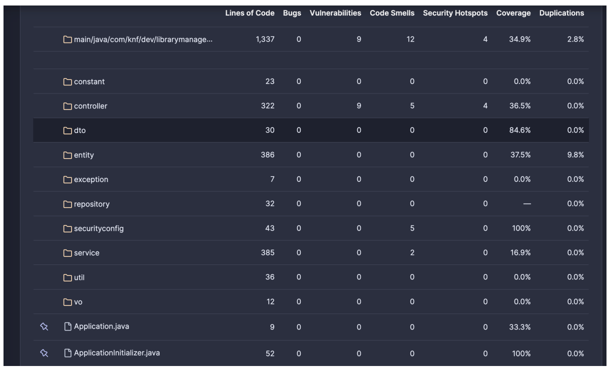
Maintainability
18 Code Smells
A

Security
13 Vulnerabilities
E

Security Review
21 Security Hotspots
0.0% Reviewed
E

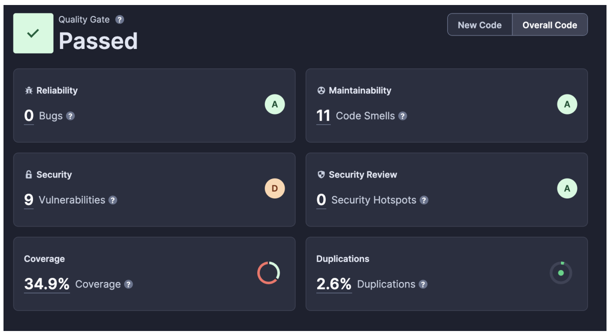
Figure 5.2: Quality Metrics of Candidate R - Library Management System before Refactoring

After Refactoring



	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
main/java/com/knf/dev/librarymanage...	1,337	0	9	12	4	34.9%	2.8%
constant	23	0	0	0	0	0.0%	0.0%
controller	322	0	9	5	4	36.5%	0.0%
dto	30	0	0	0	0	84.6%	0.0%
entity	386	0	0	0	0	37.5%	9.8%
exception	7	0	0	0	0	0.0%	0.0%
repository	32	0	0	0	0	—	0.0%
securityconfig	43	0	0	5	0	100%	0.0%
service	385	0	0	2	0	16.9%	0.0%
util	36	0	0	0	0	0.0%	0.0%
vo	12	0	0	0	0	0.0%	0.0%
Application.java	9	0	0	0	0	33.3%	0.0%
ApplicationInitializer.java	52	0	0	0	0	100%	0.0%

Figure 5.3: Package Metrics of Candidate R - Library Management System after Refactoring



Quality Gate

Passed

Reliability A
0 Bugs

Maintainability A
11 Code Smells

Security D
9 Vulnerabilities

Security Review A
0 Security Hotspots

Coverage
34.9% Coverage

Duplications
2.6% Duplications

Figure 5.4: Quality Metrics of Candidate R - Library Management System before Refactoring

6 Refactoring Challenges and Results

6.0.1 Challenges

A number of challenges were faced during the refactoring process. One of them was inadvertent breakage in functionality that could only be detected by manual testing. One example was when replacing the Author Entity in the controller with a DTO Object. Additional annotations were needed in order for the templating system to correctly identify the fields in the DTO that were previously not required for the UI to work. Lack of familiarity with the UI framework resulted in significant amount of time spent in trial and error trying various template expression and annotation combinations. Following through with the best practice recommendation lead to the amount of coupling between the UI and the entities stored in the data being reduced there by decreasing the attack surface for incoming requests with malicious intent.

6.0.2 Results

In this final report we can see the left-hand side which holds the data we retrieved when running the system through team scale initially, and the right-hand side includes after refactoring. The lines of code have gone up, as we had to add functionalities with regard to API security and Java documentation. We can also see that the finding density has gone down drastically from 88.7 to 2.9, and the finding count of undesirables has gone down by 182 undesirables. As mentioned in the previous sections, the reason we have chosen to keep these 8 undesirables is because handling them would lead to altering the functionality of the system which was not our intention for this project

7 Conclusion

For Library Management System, despite being a relatively small code base, a fairly simple application and relatively small amount a technical debt, A significant amount of effort was required to comprehend, re-engineer and test various aspects of the system. Tooling was immensely helpful in aiding our efforts. Lessons learned include the importance of unit testing and the consequences of in appropriate coding practices. A lot of the issues seen could have been avoided from the get go with frequent code scanning and analysis.

By utilizing GitHub for version control and SonarQube for code analysis, we aimed to identify and rectify any code smells and undesirables that might have been lurking in the codebase.

We uncovered various areas for improvement. SonarQube proved to be a great tool in this endeavor, providing us with valuable insights into potential bugs, duplications, and many other code smells. Armed with this knowledge, we worked collaboratively as a team to address the issues. By focusing on code quality, we have significantly improved the overall health of the librarymanagementsystem source code.

In addition to the improvements in code quality, this project has been a tremendous learning opportunity for our team. We have gained a deeper understanding of industry best practices and have embraced a culture of continuous improvement. This awareness will undoubtedly influence our approach to future in real-life projects, ensuring that we deliver software of the highest quality to our users.

In conclusion, the integration of Git, SonarQube, and Checkstyles into our development workflow has been an invaluable decision. It has empowered us to enhance code quality, improve maintainability, and provide a culture of continuous improvement. As we move forward, we are excited to apply the lessons learned from this experience to future projects.

A Tools Used to Refactor the Candidate R

1. **Eclipse IDE:** Eclipse is an integrated development environment used in computer programming. It consists of a base workspace and various plugins that allow developers to customize their environment. The debugger is a prominent feature, significantly aiding in code improvement. With its user-friendly interface, developers can easily debug, track, and navigate through different files.

2. **SonarCloud:** SonarCloud is a cloud-based platform that offers continuous code quality and security analysis. It helps developers identify and fix code issues, ensuring the delivery of clean and secure code. SonarCloud provides valuable insights into code quality, security vulnerabilities, and technical debt. By integrating with version control systems, SonarCloud offers a seamless and automated code analysis process in the cloud.

3. **Checkstyle:** Checkstyle is a development tool integrated as an IDE plugin. It automatically checks coding standards, making the software development process more efficient by indicating design problems like code formats and layout. Checkstyle aids in auditing code structure within classes and methods and also reduces the likelihood of formatting issues.

4. **Overleaf:** Overleaf is a collaborative cloud-based LaTeX editor utilized for writing, editing, and publishing scientific documents. It collaborates with various scientific publishers, providing official journal LaTeX templates and direct submission links. Overleaf facilitates teamwork, allowing multiple team members to work simultaneously on documents. Moreover, it comes equipped with pre-installed packages, streamlining the editing process.

5. **GIT:** Git is a free and open-source distributed version control system designed to handle projects of all sizes with speed and efficiency. Its easy-to-learn and lightweight nature sets it apart from SCM tools. With features such as cheap local branching and convenient staging areas, Git empowers developers with multiple workflows and enhances collaboration.

B Refactored source code of R

The candidate's R source code below refactoring and after refactoring has been updated on the below link of Git Hub.

Source Code of Candidate R after Refactoring
https://github.com/knowledgefactory4u/librarymanagementsystem

Source Code of Candidate R before Refactoring
https://github.com/skattoju4/librarymanagementsystem/tree/feature/reengineering