Concordia University
Engineering and Computer Science

SOEN 6431 - Software Comprehension And Maintenance

Deliverable 2 - Reengineering Opportunity
DÉJÀ VU

Submitted to : Dr. Pankaj Kamthan

Summarized by:
A K M Saifun Nabi(40222168)
Dhruviben Jigeshkumar Modi(40166396)
Sumit Vasharambhai Monapara(40197174)
Harshal Jigeshkumar Modi (40195060)

Github Repository Link

Contents

1. Abstract

Continuous evolution is essential for a software system's long-term market success. As part of this evolution, the original software must be continuously updated and improved in order to meet the shifting needs of the market. Software maintenance is the term used to describe this ongoing improvement process. It is crucial to carefully examine the source code and documentation and have a deep understanding of the system in order to carry out software maintenance and apply beneficial modifications.

We have chosen the R system called Billing System from the 4 candidate systems which we described in deliverable 1. In this deliverable 2, we make changes in source code R as we mentioned in deliverable 1. In this assignment, we make a report about the change in relevant software metrics before and after the re-engineering.

2. Introduction

Software re-engineering offers numerous advantages such as improved maintainability, enhanced performance, and increased flexibility. Debugging, reviewing, and testing of source code play critical roles in ensuring software reliability, identifying and resolving defects, and enhancing overall code quality. These practices contribute to the development of robust and stable software systems, leading to better user experiences and increased customer satisfaction.

The Billing System is a comprehensive store billing application developed in Java with a user-friendly graphical user interface (GUI). The system generates invoices tailored to each state's requirements. The main file, AdminPanel.java, facilitates various CRUD operations such as adding, deleting, and updating products. Additional Java files like updateProduct.java, deleteCashier.java, deleteProduct.java, showStocks.java, and addCashier.java handle specific tasks related to managing products, cashiers, and displaying stock information. Together, these files create an efficient and versatile billing system for stores.

Choosing a Java billing system project based on the team's familiarity with the language provides several benefits, including enhanced collaboration, reduced learning curve, and improved productivity. Implementing the project with the MVC design pattern ensures a proper coding structure, modularity, and maintainability. The effective utilization of Java's exception handling mechanism further enhances the system's reliability and stability. To ensure efficient operations, seamless integration with databases is necessary. This integration facilitates the storage, retrieval, and manipulation of critical data, allowing accurate billing processes, streamlined reporting, and comprehensive analysis. Proper database integration ensures data consistency, security, and scalability. This System has utilized JDBC API. The JDBC API provides a standardized set of interfaces, allowing Java applications to establish connections with various databases. It simplifies database interactions by providing methods to execute SQL queries, retrieve data, and manage transactions.

3. List of Source Code Undesirables

Our team has extensively utilized the SonarQube tool to diligently scrutinize our deliverable and detect any undesired elements present in the codebase. The findings we encountered cover a wide spectrum of undesirables, including critical issues such as use of the deprecated libraries, potential security vulnerabilities, and architectural weaknesses, as well as less critical matters such as coding style inconsistencies and suboptimal code blocks.

In this section, we present a detailed breakdown of each identified undesirable, providing comprehensive information such as its type, category, specific code smell type, and a comprehensive description. We aim to offer a profound understanding of each issue to facilitate effective resolution and enhance the overall quality of our deliverable. Throughout this meticulous process, we have meticulously documented a summary for each code smell, providing an overview of its impact on the codebase and potential implications. In total, we have identified and resolved a total of 20 findings, signifying our unwavering commitment to producing a high-quality deliverable that adheres to industry best practices and standards.

We have categorized three types of undesirables with their respective explanations: Bugs, Vulnerabilities, and Code Smells.

Bug: A bug refers to an error, flaw, or unintended behavior in the software code that leads to incorrect or unexpected results during execution.

Vulnerability: A vulnerability is 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 a piece of code that is poorly designed or written, making it difficult to maintain, understand, or extend. It doesn't necessarily lead to bugs but hinders code quality and may increase the likelihood of introducing defects.

1	Type	Bug
	Severity	Blocker
	Observation	Use try-with-resources or close this ”Statement” in a ”finally” clause.
	Code Smell Type	Resource Management - Finally block clean-up
	Code Smell Summary	Resources like connections, streams, and other classes that implement the Closeable interface should be closedin order to release system resources properly.
	Frequency	16

2	Type	Bug
	Severity	Major
	Observation	A ”NullPointerException” could be thrown; ”conn” is nullable here.
	Code Smell Type	Implementation Smells - Null Pointer Dereference.
	Code Smell Summary	Attempting to dereference or access members/methods of a reference that is null will cause a ”NullPointerException,” leading to abrupt program termination. We should always perform null checks on nullable references before accessing them.
	Frequency	15

3	Type	Vulnerability
	Severity	Blocker
	Observation	Add password protection to this database.
	Code Smell Type	Inadequate Security Measures
	Code Smell Summary	When relying on the password authentication mode for the database connection, a secure password should be chosen. This rule raises an issue when an empty password is used.
	Frequency	1

4	Type	Code Smell
	Severity	Critical
	Observation	Use static access with **javax.swing.WindowConstants** for DISPOSE_ON_CLOSE
	Code Smell Type	Static Overuse: Excessive Static Access.
	Code Smell Summary	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.
	Frequency	2

5	Type	Code Smell
	Severity	Critical
	Observation	Define a constant instead of duplicating this literal **”XXXX”** N times.
	Code Smell Type	Duplicated Variable - Magic String
	Code Smell Summary	Duplicated string literals make the process of refactoring error-prone since you must be sure to update all occurrences, but with assigning to constant, we need to update only once and access anywhere.
	Frequency	25

6	Type	Code Smell
	Severity	Critical
	Observation	Add a nested comment explaining why this method is empty or throw Exception
	Code Smell Type	Configuration Smells : Empty method
	Code Smell Summary	There are several reasons for a method not to have a method body: if the method will not be used in the future, throw an UnSupportedOperationException or if overriding it, add a nested comment explaining why this method is empty.
	Frequency	1

7	Type	Code Smell
	Severity	Critical
	Observation	Make **”XXXX”** transient or serializable.
	Code Smell Type	Implementation Smells: Serialization Smell
	Code Smell Summary	Fields in a Serializable class must themselves be either Serializable or transient,even if the class is never explicitly serialized or deserialized. This rule raises an issue on non-Serializable fields.
	Frequency	1

8	Type	Code Smell
	Severity	Critical
	Observation	Refactor this method to reduce its Cognitive Complexity from 23 to the 15 allowed.
	Code Smell Type	Complex code: High Cognitive Complexity
	Code Smell Summary	Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.
	Frequency	2

9	Type	Code Smell
	Severity	Critical
	Observation	Use static access with **javax.swing.WindowConstants** for ”EXIT_ON_CLOSE”
	Code Smell Type	Static Overuse: Excessive Static Access
	Code Smell Summary	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.
	Frequency	1

10	Type	Code Smell
	Severity	Critical
	Observation	Add a default case to this switch.
	Code Smell Type	Switch statements
	Code Smell Summary	The requirement for a final default clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken.
	Frequency	3

11	Type	Code Smell
	Severity	Major
	Observation	Replace this use of **System.out \System.err** by a **logger**.
	Code Smell Type	Implementation Smells: Logging
	Code Smell Summary	Everyone must follow the logging patterns to easily retrieve the logs for all users, the format of the log should be uniform across applications, and logged data must be recorded for future use.
	Frequency	3

12	Type	Code Smell
	Severity	Major
	Observation	Make this anonymous inner class a lambda.
	Code Smell Type	Anonymous Class Smell
	Code Smell Summary	Before Java 8, the only way to partially support closures in Java was by using anonymous inner classes, but the syntax was unclear and unwieldy, so after Java 8, anonymous inner classes should be replaced by lambdas to highly increase the readability.
	Frequency	22

13	Type	Code Smell
	Severity	Major
	Observation	This block of commented-out lines of code should be removed.
	Code Smell Type	Implementation Smells - Comments
	Code Smell Summary	Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required
	Frequency	7

14	Type	Code Smell
	Severity	Major
	Observation	Remove this useless assignment to local variable **”XXXX”**.
	Code Smell Type	Resource Management - Dead Store
	Code Smell Summary	dead store happens when a local variable is assigned a value that is not read by any subsequent instruction and leads to waste of storage, Therefore all calculated values should be used.
	Frequency	4

15	Type	Code Smell
	Severity	Minor
	Observation	Rename this class name to match the regular expression ’^[A-Z][a-zA-Z0-9]*$’..
	Code Smell Type	Configuration Smells : Inconsistent Naming Convention
	Code Smell Summary	Shared coding conventions allow teams to collaborate effectively. This rule allows us to check that all class names match a provided regular expression..
	Frequency	9

16	Type	Code Smell
	Severity	Minor
	Observation	Remove this unused import **”AB.CD.EF”**.
	Code Smell Type	Architectural Smells - Unused Packages
	Code Smell Summary	The import part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Leaving them in reduces the code’s readability since their presence can be confusing.
	Frequency	28

17	Type	Code Smell
	Severity	Info
	Observation	Complete the task associated to this TODO comment..
	Code Smell Type	Implementation Smells : Comments
	Code Smell Summary	**TODO** tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. This rule is meant to track those tags and to ensure that they do not go unnoticed.
	Frequency	26

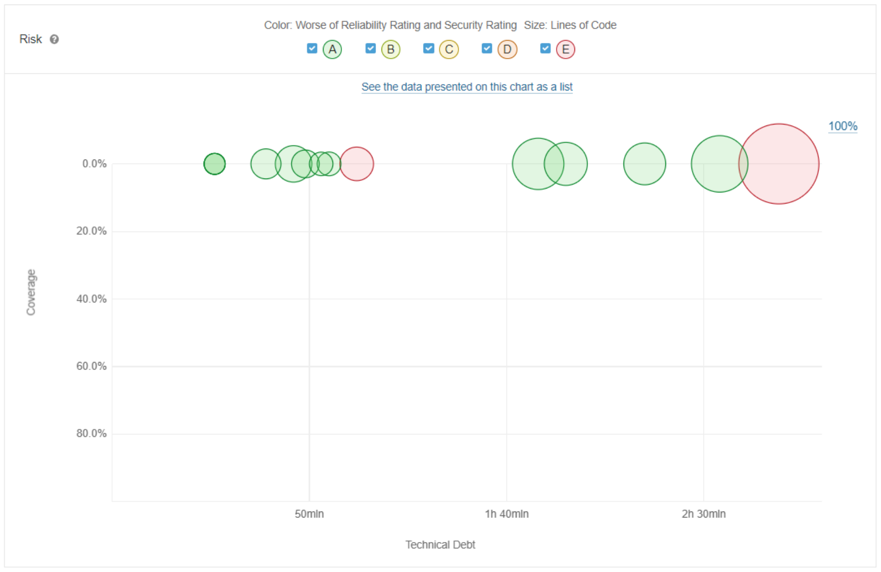
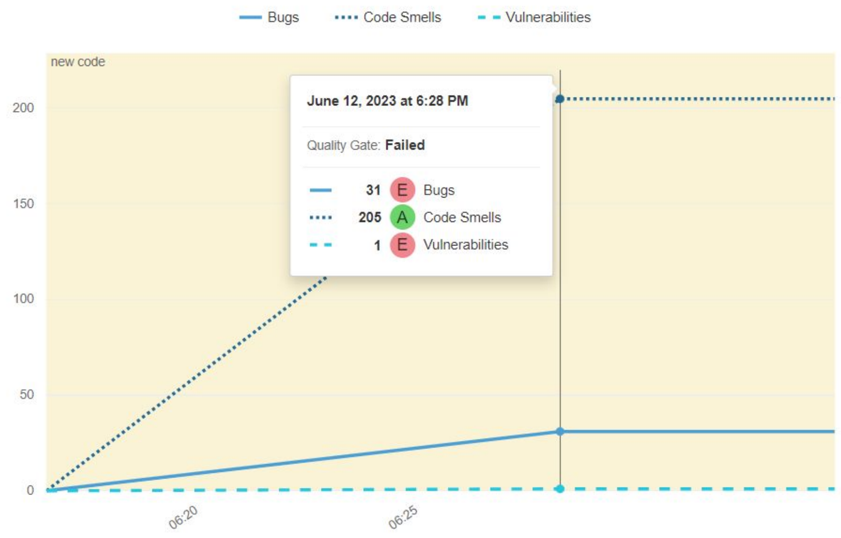


Figure 1: SonarQube Analysis of issues per Java class (Billing System)



June 12, 2023 at 6:28 PM

Quality Gate: **Failed**

31 E Bugs
205 A Code Smells
1 E Vulnerabilities

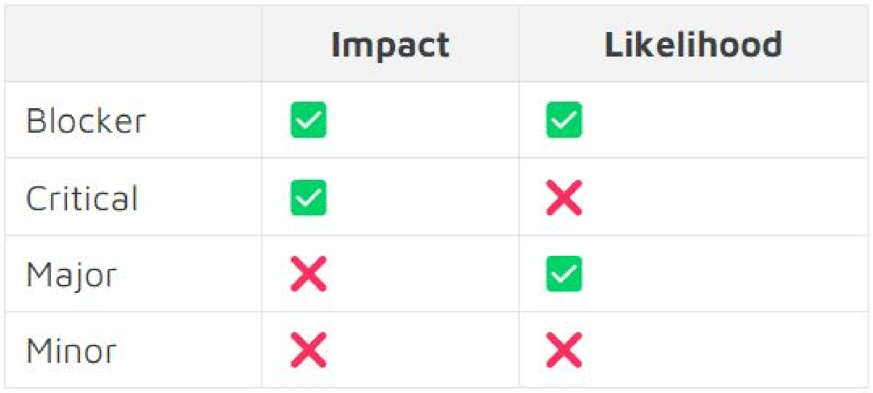
Figure 2: SonarQube Analysis of Each undesirables (Billing System)

4. Re-engineering Methods for Undesrirables

In this particular section, we have provided a comprehensive account of the mapping process between source code undesirables and the corresponding re-engineering methods. Our aim was to establish a clear association between the observed undesirables and the appropriate re-engineering rule, rule type, issue tag, undesirable severity, undesirable likelihood, technical debt.

Since our chosen system is implemented in Java, we employed Java Static Code Analysis as a valuable tool for this endeavor. Relying on the insights gained from this analysis, we were able to accurately identify and categorize the aforementioned elements, ensuring a systematic and methodical approach to the evaluation.

By leveraging the insights derived from the Java Static Code Analysis graph, we could confidently pinpoint and assess the significance of each mapping, ultimately leading us to make informed decisions on how best to address the identified undesirables and implement the suitable reengineering methods. This approach fosters a well-organized and targeted strategy for optimizing the system's code quality and overall performance.



	Impact	Likelihood
Blocker	✓	✓
Critical	✓	✗
Major	✗	✓
Minor	✗	✗

Figure 3: SonarQube Analysis of Severity Category

1	Re-engineering Rule Type	Bug
	Severity	Blocker
	Likelihood	High
	Observation	Use try-with-resources or close this ”Statement” in a ”finally” clause.
	Re-engineering Rule Description	Connections, streams, files, and other classes that implement the Closeable interface or its super-interface, AutoCloseable, needs to be closed after use. Further, that close call must be made in a finally block otherwise an exception could keep the call from being made. Preferably, when a class implements AutoCloseable, the ”try-with-resources” pattern should be used to ensure automatic resource cleanup
	Issue Tag	cwe-459, bad practice, cert - FIO04-J
	Technical Debt	80 min effort
	Refactored By	Sumit Monapara

2	Re-engineering Rule Type	Bug
	Severity	Major
	Likelihood	High
	Observation	A ”NullPointerException” could be thrown; ”conn” is nullable here.
	Re-engineering Rule Description	A reference to null should never be dereferenced/accessed. Doing so will cause a NullPointerException to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures.
	Issue Tag	cwe-476, cert - EXP01-J, bad practice
	Technical Debt	120 min effort
	Refactored By	Sumit Monapara

3	Re-engineering Rule Type	Vulnerability
	Severity	Blocker
	Likelihood	Rare
	Observation	Add password protection to this database.
	Re-engineering Rule Description	When utilizing password authentication for the database connection, it is crucial to select a secure password. An issue arises when an empty password is employed.
	Issue Tag	cwe-521, OWASP-A3
	Technical Debt	45 min effort
	Refactored By	Sumit Monapara

4	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	Use static access with **javax.swing.WindowConstants** for DISPOSE_ON_CLOSE
	Re-engineering Rule Description	To maintain code clarity, avoid accessing static members of a base class using the name of a derived type. This practice can lead to confusion and may wrongly imply the existence of two distinct static members.
	Issue Tag	confusing
	Technical Debt	5 min effort
	Refactored By	Harshal Modi

5	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	Define a constant instead of duplicating this literal **”XXXX”** N times.
	Re-engineering Rule Description	By duplicating Literals, we 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.
	Issue Tag	design, duplicate
	Technical Debt	225 min effort
	Refactored By	Sumit Monapara

6	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	Add a nested comment explaining why this method is empty or throw Exception.
	Re-engineering Rule Description	There are many reasons to have a method without a body: if the method will not be used in the future, throw an UnSupportedOperationException or if overriding it, add a nested comment explaining why this method is empty.
	Issue Tag	suspicious
	Technical Debt	5 min effort
	Refactored By	Harshal Modi

7	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	**"XXXX"** transient or serializable.
	Re-engineering Rule Description	In a Serializable class, all fields must either be Serializable or marked as transient, even if the class is never explicitly serialized or deserialized. This rule highlights potential issues with non-Serializable fields and with collection fields that are not private and are assigned non-Serializable types within the class.
	Issue Tag	cwe-594, serialization
	Technical Debt	30 min effort
	Refactored By	Harshal Modi

8	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	Refactor this method to reduce its Cognitive Complexity from 23 to the 15 allowed.
	Re-engineering Rule Description	Cognitive Complexity is a measurement of the difficulty in understanding the control flow of a method. Methods with high Cognitive Complexity will be challenging to maintain due to their intricate control structures, making them hard to comprehend and modify.
	Issue Tag	brain-overload
	Technical Debt	21 min effort
	Refactored By	Harshal Modi

9	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	Rare
	Observation	Use static access with **javax.swing.WindowConstants** for "EXIT_ON_CLOSE"
	Re-engineering Rule Description	To maintain code clarity, avoid accessing static members of a base class using the name of a derived type. This practice can lead to confusion and may wrongly imply the existence of two distinct static members.
	Issue Tag	confusing
	Technical Debt	5 min effort
	Refactored By	Harshal Modi

10	Re-engineering Rule Type	Code Smell
	Severity	Critical
	Likelihood	High
	Observation	Add a default case to this switch.
	Re-engineering Rule Description	If the switch parameter is an Enum and if all the constants of this enum are used in the case statements, then no default clause is expected. In other cases, we must need a default case to catch the additional conditions that are not caught in other catch blocks.
	Issue Tag	cwe-478, cert-MSC01-C
	Technical Debt	15 min effort
	Refactored By	Dhruvi Modi

11	Re-engineering Rule Type	Code Smell
	Severity	Major
	Likelihood	Rare
	Observation	Replace this use of **System.out \System.err** by a **logger**.
	Re-engineering Rule Description	Logging messages must meet the below requirements: Users should have convenient access to retrieve the logs. All logged messages must have a consistent format for easy readability. The logged data must be reliably recorded. Sensitive data should only be logged in a secure manner. To fulfill these requirements, it is highly recommended to define and use a dedicated logger instead of directly writing to standard outputs in a program.
	Issue Tag	OWASP-A9, bad practice, cert-ERR02-J
	Technical Debt	30 min effort
	Refactored By	Dhruvi Modi

12	Re-engineering Rule Type	Code Smell
	Severity	Major
	Likelihood	High
	Observation	Make this anonymous inner class a lambda.
	Re-engineering Rule Description	Prior to Java 8, the only method to achieve partial support for closures in Java was through anonymous inner classes. However, the syntax of anonymous classes could be cumbersome and unclear. With the introduction of Java 8, the use of lambdas significantly improved the readability of the source code.
	Issue Tag	version greater than java8
	Technical Debt	110 min effort
	Refactored By	Dhruvi Modi

13	Re-engineering Rule Type	Code Smell
	Severity	Major
	Likelihood	Rare
	Observation	This block of commented-out lines of code should be removed.
	Re-engineering Rule Description	Unused code should be deleted and can be retrieved from source control history if required.
	Issue Tag	unused
	Technical Debt	35 min effort
	Refactored By	Dhruvi Modi

14	Re-engineering Rule Type	Code Smell
	Severity	Major
	Likelihood	Rare
	Observation	Remove this useless assignment to local variable **”XXXX”**.
	Re-engineering Rule Description	A dead store occurs when a local variable is assigned a value that is not utilized by any subsequent instruction. This situation, where a value is calculated or retrieved but then immediately overwritten or discarded, may indicate a significant coding error. As a result, it is crucial to utilize all calculated values to avoid dead stores in the code.
	Issue Tag	cwe-563, cert-MSC13-C, unused
	Technical Debt	60 min effort
	Refactored By	A K M Saifun Nabi

15	Re-engineering Rule Type	Code Smell
	Severity	Minor
	Likelihood	Rare
	Observation	Rename this class name to match the regular expression ’[̂A-Z][a-zA-Z0-9]*$’..
	Re-engineering Rule Description	Shared naming conventions allow teams to collaborate efficiently. This rule checks that all method names match a provided regular expression.
	Issue Tag	convention
	Technical Debt	45 min effort
	Refactored By	A K M Saifun Nabi

16	Re-engineering Rule Type	Code Smell
	Severity	Minor
	Likelihood	Rare
	Observation	Remove this unused import **”AB.CD.EF”**.
	Re-engineering Rule Description	The import part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Unused and useless imports should not occur if that is the case. Leaving them in reduces the code’s readability since their presence can be confusing.
	Issue Tag	unused
	Technical Debt	56 min effort
	Refactored By	A K M Saifun Nabi

17	Re-engineering Rule Type	Code Smell
	Severity	Info
	Likelihood	Rare
	Observation	Complete the task associated to this TODO comment..
	Re-engineering Rule Description	TODO tags are frequently utilized to indicate areas in the code that require additional implementation but are intended to be addressed later by the developer. However, there is a risk that the developer might forget or not have the time to revisit those tags. This rule is designed to monitor and ensure that these TODO tags are not overlooked or left unaddressed.
	Issue Tag	cwe-546
	Technical Debt	26 min effort
	Refactored By	A K M Saifun Nabi

5. Occurrences of Source Code Undesirables

This tabulated data presents a comprehensive summary of the identified undesirables, outlining their specific occurrences within the codebase. Furthermore, the number of occurrences of different undesirables per file is included, providing a detailed overview of the distribution and prevalence of these undesirables throughout the codebase.

No.	**File**	**Lines****	**Lines of code***	**Bug**	**Vulnerability**	**Code Smell**
1	src\addCashier.java	81	66	0	0	11
2	src\addProduct.java	112	94	0	0	8
3	src\AdminPanel.java	220	189	0	0	15
4	src\DB.java	359	317	30	1	36
5	src\deleteCashier.java	80	65	0	0	10
6	src\deleteProduct.java	69	54	0	0	6
7	src\generateInvoice.java	109	84	0	0	9
8	src\Invoice.java	250	212	0	0	21
9	src\Login.java	198	152	0	0	25
10	src\Sale.java	179	147	0	0	24
11	src\searchCashier.java	69	54	0	0	6
12	src\searchProduct.java	69	54	0	0	6
13	src\showStock.java	132	110	1	0	15
14	src\updateProduct.java	149	123	0	0	13
		LOC : 2076	**SLOC : 1721**	**31**	**1**	**205**

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

Upon examining the table presented above, it becomes evident that the entire codebase consists of **2076 Lines of Code (LOC) with 1721 Source Lines of Code (SLOC)**. Within this substantial codebase, **a total of 237 undesirables** were identified, encompassing diverse issues such as bugs, vulnerabilities, and code smells. This comprehensive evaluation provides valuable insights into the overall quality and potential areas of improvement within the software, highlighting the significance of addressing these undesirables to enhance the system's reliability and security.

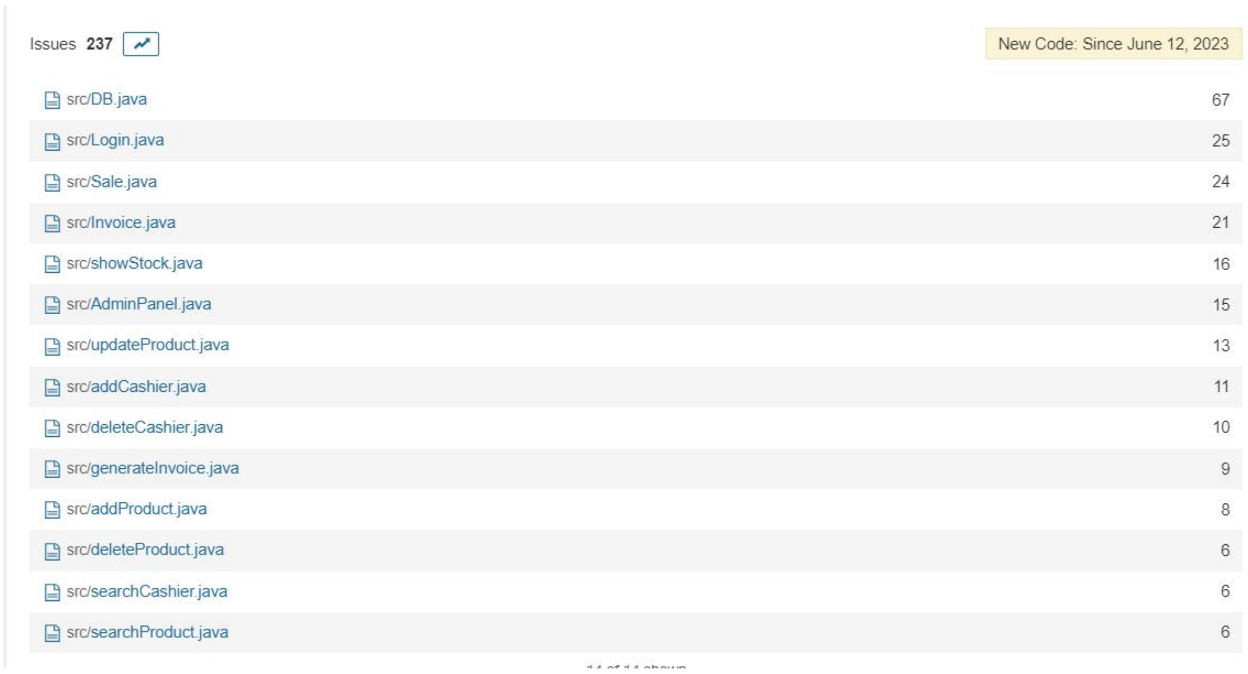
Upon the discovery of undesirable elements in the system, diligent efforts were made to address and rectify them effectively. As a result, the number of undesirables was impressively reduced from **237 down to a mere 19**. Among these 19 remaining issues, it is worth noting that they primarily consist of bugs. Nevertheless, the team successfully managed to resolve an impressive count of 205 code smells, 1 vulnerability, and 12 additional bugs.

6. Re-engineering Software Metrics Analysis

A Software Metric Log is a record or collection of various software metrics that are used to quantify different aspects of software development and maintenance. Software metrics are measurements or indicators that provide insight into the quality, complexity, size, maintainability, and other characteristics of the software system.

Monitoring and maintaining a Software Metric Log is an essential practice to track the evolution of the codebase's quality over time and to identify areas that may need further improvement. In relation to the Candidate System R, which is the Billing System, an evaluation was conducted using SonarQube, a sophisticated tool for analyzing software metrics. The primary objective of this evaluation was to identify any variations in the system's quality before and after the refactoring process. The ensuing representation focuses on the pertinent changes observed in the software metrics of the Candidate R, illustrating the advancements achieved through the process of refactoring.

Before Refactoring

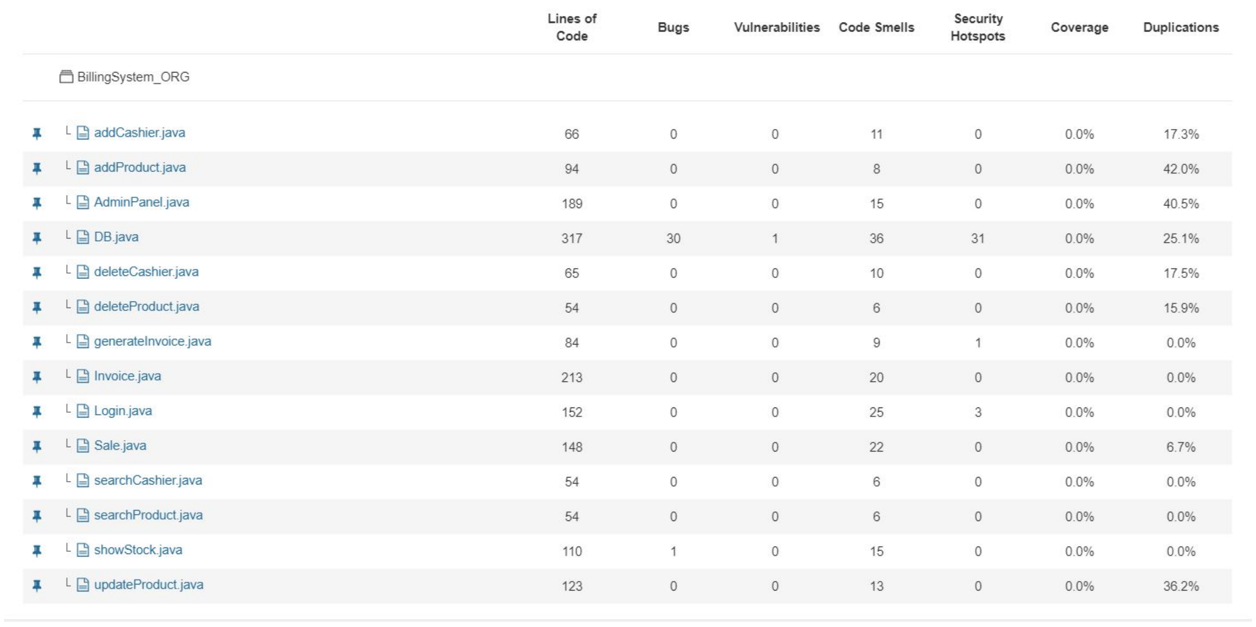


Issues 237

New Code: Since June 12, 2023

src/DB.java	67
src/Login.java	25
src/Sale.java	24
src/Invoice.java	21
src/showStock.java	16
src/AdminPanel.java	15
src/updateProduct.java	13
src/addCashier.java	11
src/deleteCashier.java	10
src/generateInvoice.java	9
src/addProduct.java	8
src/deleteProduct.java	6
src/searchCashier.java	6
src/searchProduct.java	6

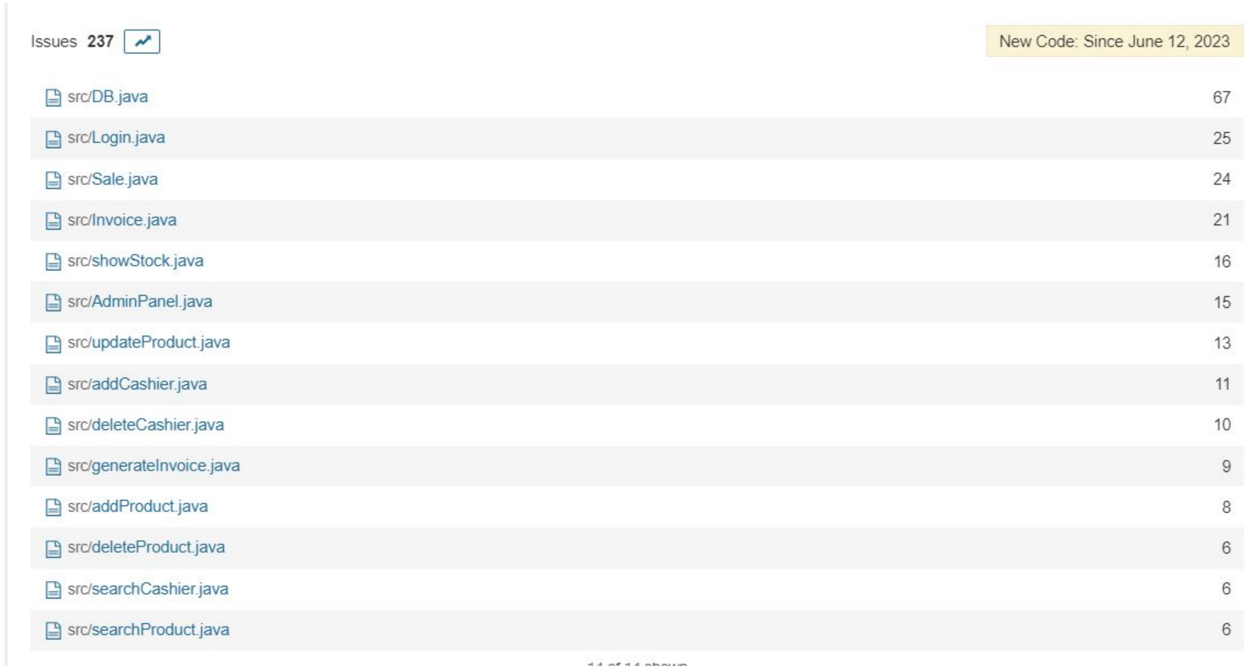
Figure 4: Before Refactoring - Total Findings (Billing System)



	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
BillingSystem_ORG							
addCashier.java	66	0	0	11	0	0.0%	17.3%
addProduct.java	94	0	0	8	0	0.0%	42.0%
AdminPanel.java	189	0	0	15	0	0.0%	40.5%
DB.java	317	30	1	36	31	0.0%	25.1%
deleteCashier.java	65	0	0	10	0	0.0%	17.5%
deleteProduct.java	54	0	0	6	0	0.0%	15.9%
generateInvoice.java	84	0	0	9	1	0.0%	0.0%
Invoice.java	213	0	0	20	0	0.0%	0.0%
Login.java	152	0	0	25	3	0.0%	0.0%
Sale.java	148	0	0	22	0	0.0%	6.7%
searchCashier.java	54	0	0	6	0	0.0%	0.0%
searchProduct.java	54	0	0	6	0	0.0%	0.0%
showStock.java	110	1	0	15	0	0.0%	0.0%
updateProduct.java	123	0	0	13	0	0.0%	36.2%

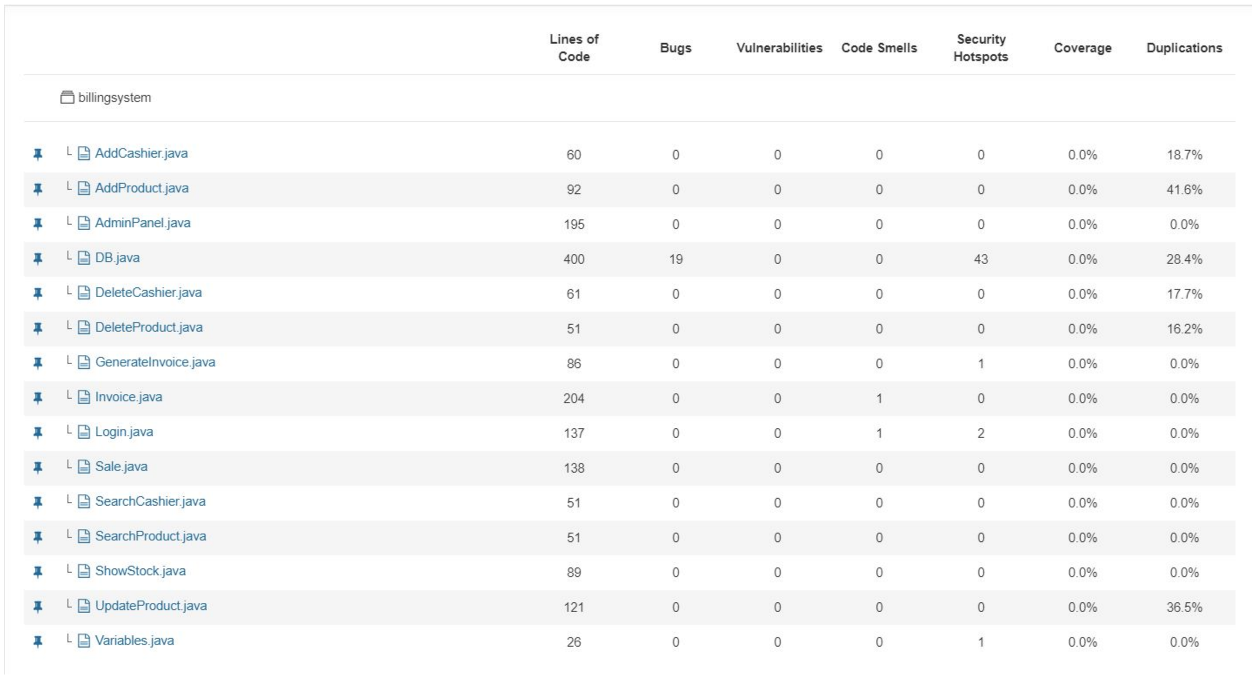
Figure 5: Before Refactoring - Quality Metrics (Billing System)

After Refactoring



Issues 237	New Code: Since June 12, 2023
src/DB.java	67
src/Login.java	25
src/Sale.java	24
src/Invoice.java	21
src/showStock.java	16
src/AdminPanel.java	15
src/updateProduct.java	13
src/addCashier.java	11
src/deleteCashier.java	10
src/generateInvoice.java	9
src/addProduct.java	8
src/deleteProduct.java	6
src/searchCashier.java	6
src/searchProduct.java	6

Figure 6: After Refactoring - Total Findings (Billing System)



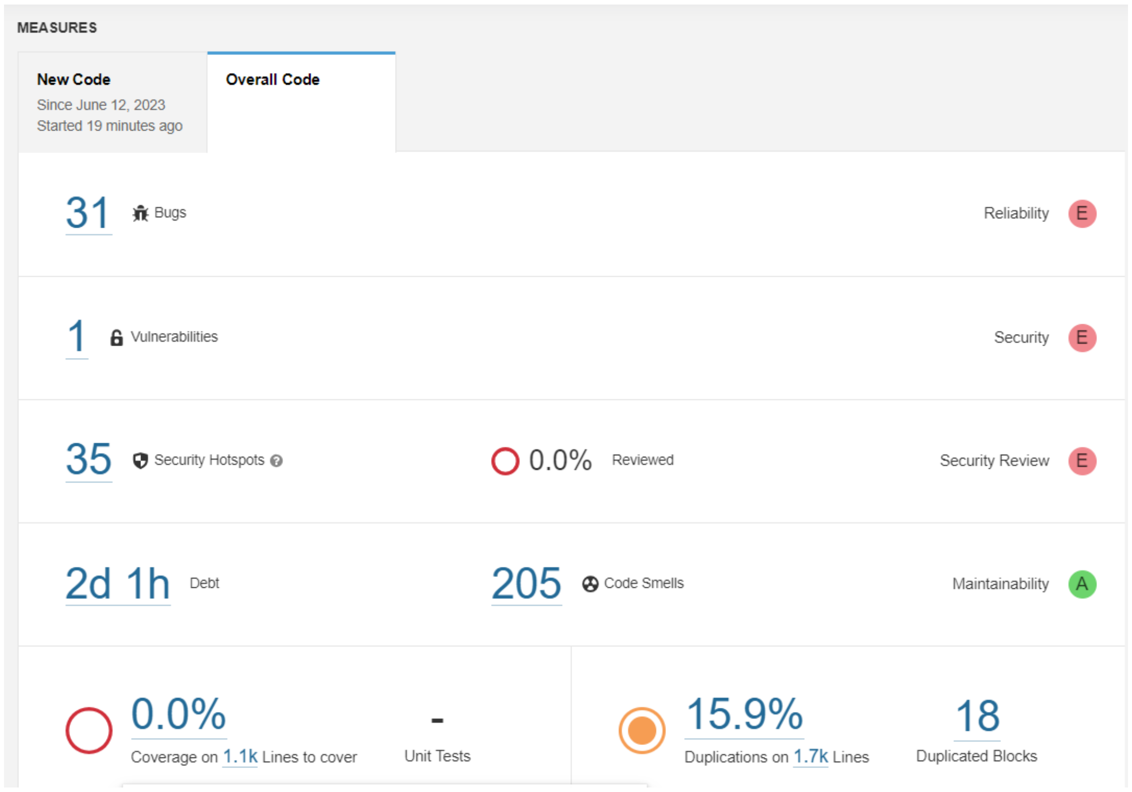
	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
billingsystem							
AddCashier.java	60	0	0	0	0	0.0%	18.7%
AddProduct.java	92	0	0	0	0	0.0%	41.6%
AdminPanel.java	195	0	0	0	0	0.0%	0.0%
DB.java	400	19	0	0	43	0.0%	28.4%
DeleteCashier.java	61	0	0	0	0	0.0%	17.7%
DeleteProduct.java	51	0	0	0	0	0.0%	16.2%
GenerateInvoice.java	86	0	0	0	1	0.0%	0.0%
Invoice.java	204	0	0	1	0	0.0%	0.0%
Login.java	137	0	0	1	2	0.0%	0.0%
Sale.java	138	0	0	0	0	0.0%	0.0%
SearchCashier.java	51	0	0	0	0	0.0%	0.0%
SearchProduct.java	51	0	0	0	0	0.0%	0.0%
ShowStock.java	89	0	0	0	0	0.0%	0.0%
UpdateProduct.java	121	0	0	0	0	0.0%	36.5%
Variables.java	26	0	0	0	1	0.0%	0.0%

Figure 7: After Refactoring - Quality Metrics (Billing System)

7. Refactoring Report

In this comprehensive analysis, we present a detailed comparison between the state of the code before and after the refactoring process, taking into account various aspects such as technical debt and maintainability. By meticulously examining the codebase in both its pre-refactoring and post-refactoring states, we aim to shed light on the significant improvements achieved and the measures taken to address technical debt and enhance the overall maintainability of the software.

The decision to retain these 19 undesirables out of the initial 237 was taken with careful consideration, as altering them could potentially lead to changes in the system's core functionality as these 19 issues are false positive. Therefore, the focus was on prioritizing the issues based on their impact and criticality, ensuring that the essential functionalities remained intact while simultaneously improving the overall quality and performance of the system.


MEASURES

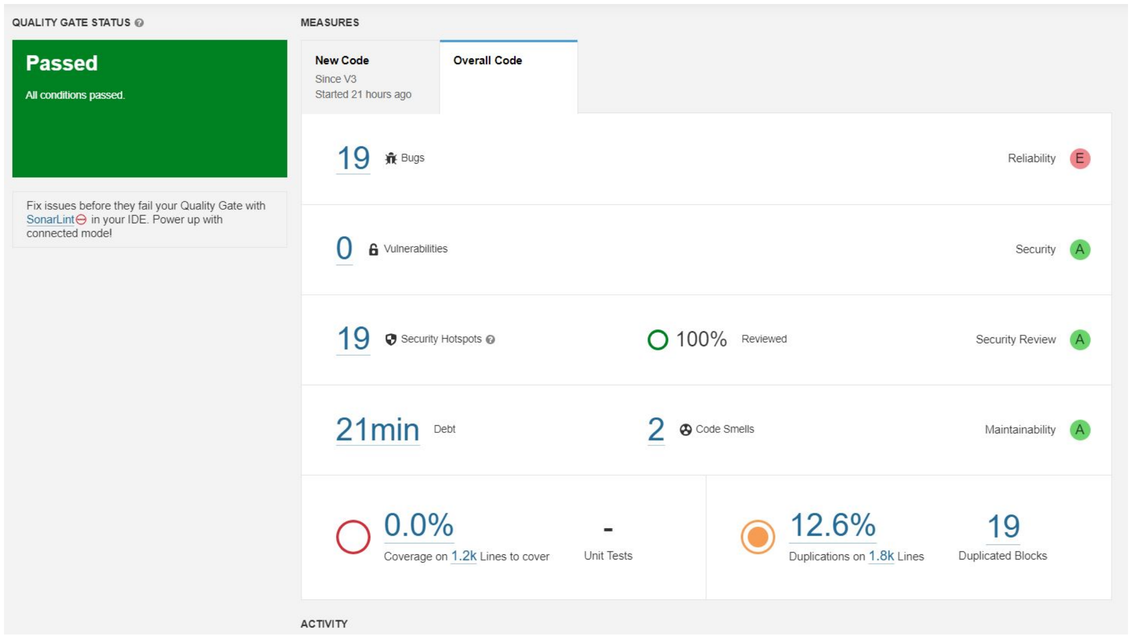
New Code
Since June 12, 2023
Started 19 minutes ago

Overall Code

31 Bugs		Reliability E
1 Vulnerabilities		Security E
35 Security Hotspots	0.0% Reviewed	Security Review E
2d 1h Debt	205 Code Smells	Maintainability A

0.0% Coverage on 1.1k Lines to cover	- Unit Tests	15.9% Duplications on 1.7k Lines	18 Duplicated Blocks

Figure 8: SonarQube Report - Before Refactoring


QUALITY GATE STATUS

Passed
All conditions passed.

Fix issues before they fail your Quality Gate with SonarLint in your IDE. Power up with connected mode!

MEASURES

New Code
Since V3
Started 21 hours ago

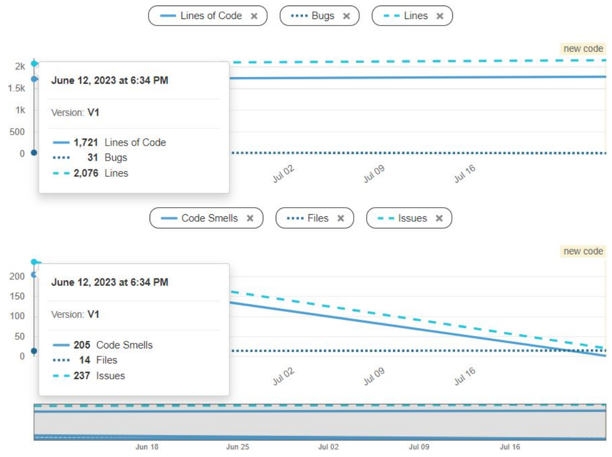
Overall Code

19 Bugs		Reliability E
0 Vulnerabilities		Security A
19 Security Hotspots	100% Reviewed	Security Review A
21min Debt	2 Code Smells	Maintainability A

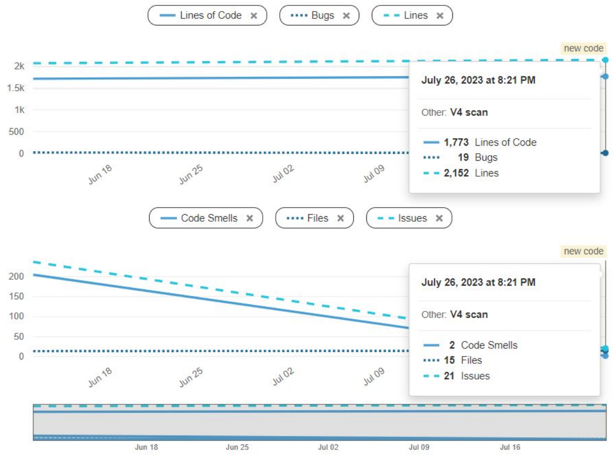
0.0% Coverage on 1.2k Lines to cover	- Unit Tests	12.6% Duplications on 1.8k Lines	19 Duplicated Blocks

ACTIVITY

Figure 9: SonarQube Report - After Refactoring

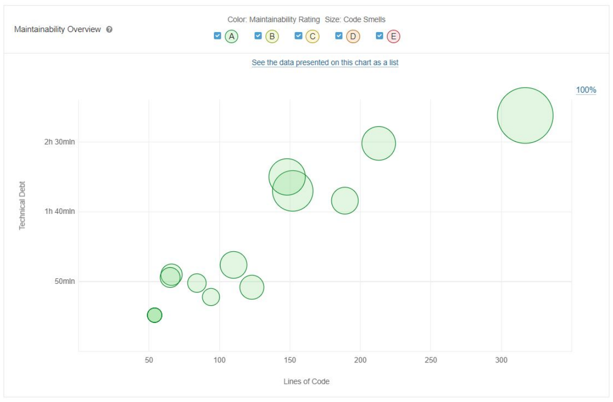


(a) Before Refactoring

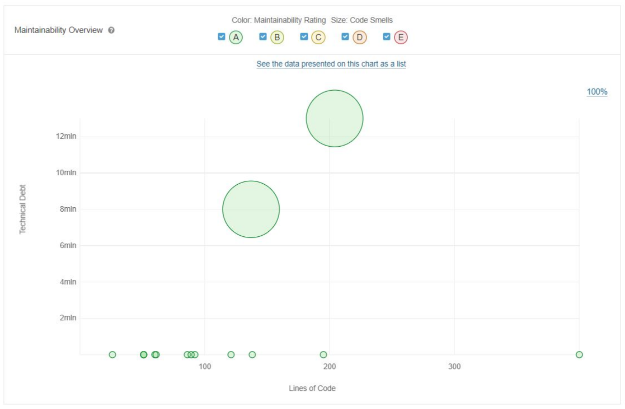


(b) After Refactoring

Figure 10: Comparison of Custom Metrics

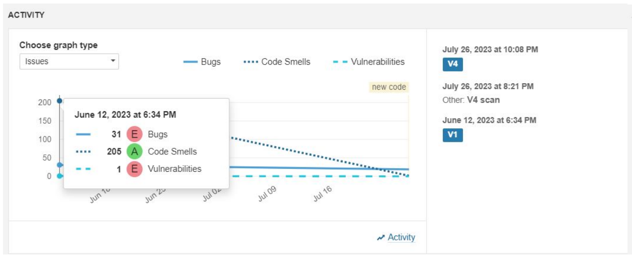


(a) Before Refactoring

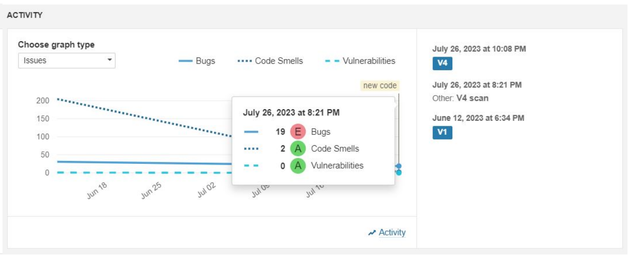


(b) After Refactoring

Figure 11: Comparison of Maintainability

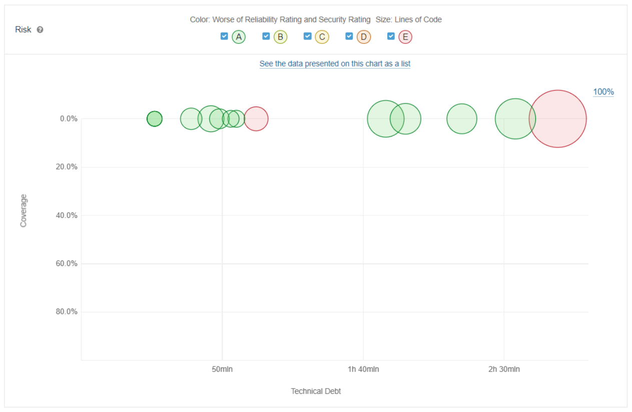


(a) Before Refactoring

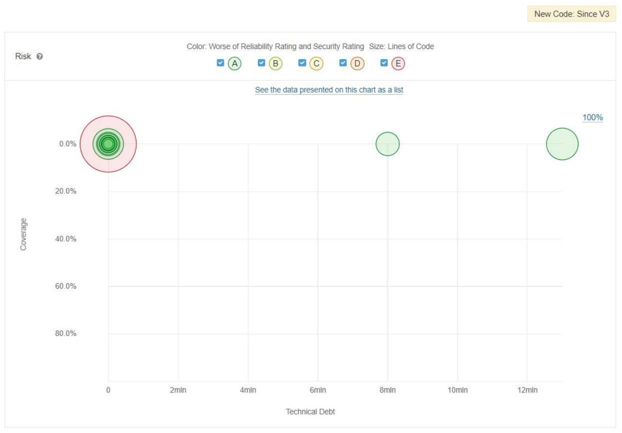


(b) After Refactoring

Figure 12: Comparison of Activity Graphs



(a) Before Refactoring



(b) After Refactoring

Figure 13: Comparison of Project Overview

8. Tools Used to Refactor the Candidate R

Below are the important tools we used to refactor Candidate R:

- **Eclipse IDE**: The Eclipse IDE (Integrated Development Environment) is a popular software development tool that provides a rich set of features for coding, debugging, and testing applications. It is the second-most-popular IDE for Java development. It provides a powerful code editor with intelligent features like code completion, syntax highlighting, and code refactoring, which help increase productivity and maintain code quality. We used this for our code refactoring and its debugger helped us to improve the code quality in many ways.
- **Overleaf**: Overleaf is an online platform for collaborative writing and editing of documents, particularly focused on scientific and technical papers. It provides a user-friendly interface where writers can easily create and format their documents using LaTeX, a typesetting system commonly used for technical and scientific publications. We used overleaf collaboration feature so that we can work on the project document simultaneously and see each others edit in real time.
- **GIT**: GIT, short for ”Global Information Tracker,” is a version control system that is used to track changes in files and directories. It allows multiple users to collaborate on a project by providing a way to manage and merge changes made by different individuals. We used GIT to organize our work and also reduce conflict during development. It also helped us to track changes made by different team members and provide facilities during merging code.
- **SonarQube**: SonarQube is an open-source platform used for continuous code quality inspection. We used this tool to detect issues related to code duplication, coding standards,

security vulnerabilities, and bugs, among others. It also provides us with a range of metrics, including code coverage, complexity, and technical debt, which helps in tracking the progress of code quality improvement efforts.

- **Teamscale**: Teamscale is a software quality management platform that helps teams ensure the reliability and maintainability of their software projects. We analyzed code repositories using this and it provided detailed insights into various code quality metrics such as code complexity, duplications, and coding standards violations. This helped us to identify areas that need improvement and to take appropriate actions to resolve potential issues.

- **VS Code**: VS Code, short for Visual Studio Code,is lightweight, efficient, and highly customizable code editor developed by Microsoft. It has very user-friendly interface and versatile features along with rich ecosystem of extensions. We used VS code for pair programming, debugging and code-refactoring.

9. Source Code of Candidate System R

In this particular segment, we have included the Source Code of the Candidate System R both prior to and following the completion of the Refactoring process.

Source Code of Candidate R (Billing System) Before Refactoring

Source Code of Candidate R (Billing System) After Refactoring

10. References

1. ISO/IEC 25010 Standard [Online].
2. Software Reengineering Strategies [Online].
3. Teamscale Integration for Visual Studio [Online].
4. VSCode Live Share : A Great Way to Collaborate with Your Team [Online].
5. SonarQube 10.1 Documentation [Online].
6. Teamscale Documentation [Online].
7. Satya Sobhan Panigrahi, Pappu Sharada, Dr,Sujit Panda (2016). Software Engineering: Re-Engineering Metrics. International Journal of Engineering Research and Application, Vol. 6, Issue 9, pp.84-87.
8. GIT Documentation [Online]