SOEN 6431
Software Comprehension and Maintenance

Varun Pandey
Preet Angad Singh Nanda
Mir Pasad
Mahavir Patel

Deliverable 2
Re-engineering Operationalization

https://github.com/DEJA-VU—SOEN-6431-SCM

Contents

1 Abstract

This document discusses the outcomes of a software re-engineering project aimed at improving a Java-based employee payroll system. The project sought to increase code maintainability and system future-proofing. This solution was chosen due to the team's comfort and the code's conformance to project criteria. To detect separate flaws in the code, two code analysis tools, Teamscale and SonarLint, were used. Teamscale was picked for its full analysis, which provided severity levels, precise locations, and potential solutions to the problems. Several undesired components, such as empty blocks, incorrect variable naming, and missing braces, were effectively rectified by the team. The efforts of re-engineering resulted in a well-structured codebase, ensuring the system's lifespan and improved functionality, allowing efficient tracking of remuneration and personnel information.

2 Introduction

Software Re-engineering is the process of making changes to a source code to make the code more maintainable and future proof. We have selected an employee payroll system for this project as mentioned in deliverable 1.

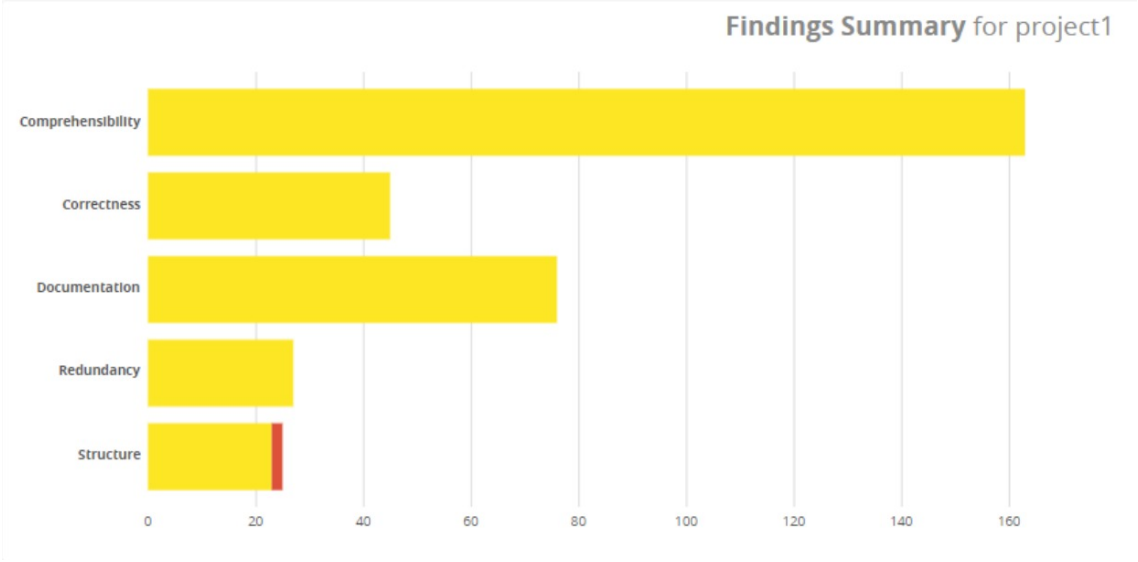
The employee payroll system is programmed in Java and later the author has compiled an executable jar for this program. This is used to keep a track of the compensation and the employed department of each and every employee in the company. Once the project is executed we need to login using out admin credentials. For there we can view all the employees along with their relevant information. From there we can navigate to other windows where we can add new employees, edit existing employees, add and edit department information and add and edit admin credentials.

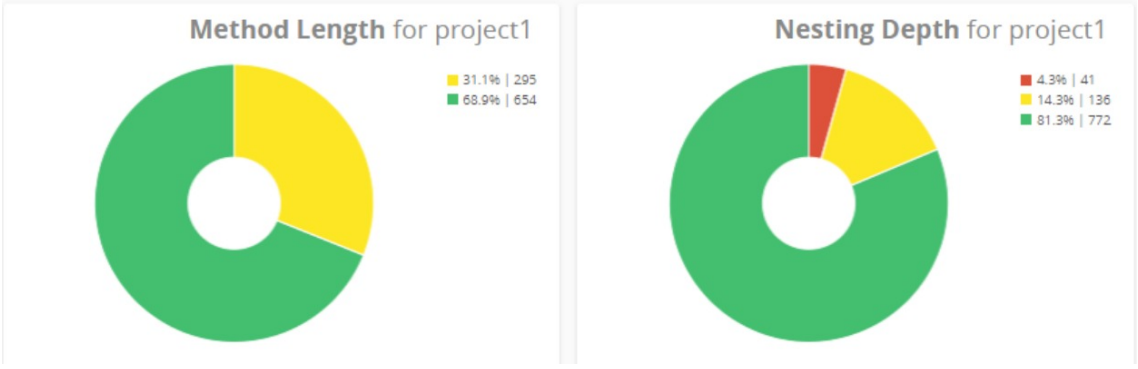
The reason we chose this system is because this was the project all of us were most comfortable with amongst all the options. More detailed reasons can be found in deliverable 1. There were additional reason, such as the source code of R satisfied all the requirement as mentioned in the project description. The code was well written and structured, and therefore was it was easier for the team to read and understand the system. We chose 2 different software to audit our code, they are Teamscale and sonar-lint. The reason we chose 2 software is that we can later compare and contrast the results obtained from them and write our insights based on that. We then went through the code and each teammates was able to point out distinct issue with the code and later went on the fix those issues.

The software re-engineering project focused on enhancing the Java-based employee payroll system has been successful. The system's selection was driven by the team's comfort and the code's adherence to project requirements. Through thorough code auditing using Teamscale and SonarLint, distinct issues were identified and effectively addressed. The re-engineering efforts have improved the system's maintainability and future-proofed its functionality. With a well-structured codebase, the payroll system now efficiently tracks compensation and employee department information, offering a user-friendly experience for administrators. Overall, the project's accomplishments have ensured the system's longevity and enhanced its performance.

3 Undesirable summary

As mentioned in the introduction we used teamscale and sonar lint to audit the code, but after looking at the output we got from bother, we decided to go with teamscale's analysis because the teamscale was giving us more information about the errors, like severity level, location and possible fixes. Apart from that, teamscale also gave us better visuals for the issues.





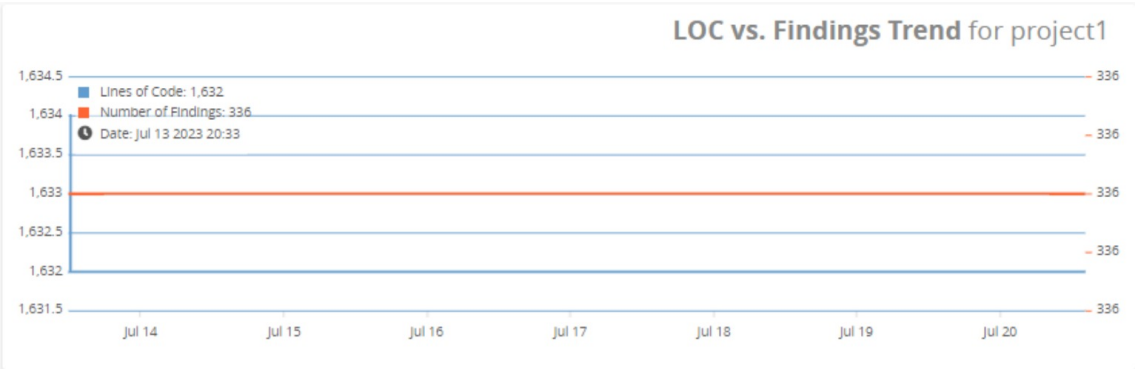


Figure 1: Lines of Code vs. Findings Trend of Candidate R - Employee Payroll System

1	Error Message	Empty block: method
	Occurrence	3
	Type	Bad Practice/case-empty-block
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Empty Catch Block

2	Error Message	Static variable dbManager is not initialized
	Occurrence	1
	Type	Error-prone Practices
	Category	Correctness
	Severity	Yellow
	Code Smell	Code Quality

3	Error Message	Attribute ‘btn_delete‘ violates naming convention.*‘
	Occurrence	7
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

4	Error Message	Attribute ‘btn_exit‘ violates naming convention.*‘
	Occurrence	1
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

5	Error Message	Non-constant public attribute dbManager
	Occurrence	1
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Code quality

6	Error Message	Attribute ‘btn_OK‘ violates naming convention.*‘
	Occurrence	1
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

7	Error Message	Attribute ‘btn_update‘ violates naming convention.*‘
	Occurrence	1
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

8	Error Message	Method System.exit should not be called
	Occurrence	2
	Type	Discouraged APIs
	Category	Correctness
	Severity	Yellow
	Code Smell	Bug

9	Error Message	Attribute ‘lbl_basic_salary‘ violates naming convention.*‘
	Occurrence	2
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

10	Error Message	Attribute ‘lbl_da‘ violates naming convention.*‘
	Occurrence	2
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

11	Error Message	Attribute ‘lbl_dep_name‘ violates naming convention.*‘
	Occurrence	3
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

12	Error Message	Attribute ‘lbl_department‘ violates naming convention.*‘
	Occurrence	2
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

13	Error Message	Attribute ‘lbl_em‘ violates naming convention.*‘
	Occurrence	2
	Type	Naming/JAVA
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Inconsistent Naming

14	Error Message	Clone with 2 instances of length x
	Occurrence	27
	Type	Redundancy/Clones
	Category	Redundancy
	Severity	Yellow
	Code Smell	Insufficient Abstraction

15	Error Message	else statement without braces
	Occurrence	8
	Type	Error-prone Practices
	Category	Correctness
	Severity	Yellow
	Code Smell	Missing Braces

16	Error Message	if statement without braces
	Occurrence	13
	Type	Error-prone Practices
	Category	Correctness
	Severity	Yellow
	Code Smell	Missing Braces

17	Error Message	Interface comment missing
	Occurrence	75
	Type	Comments/Missing Interface Comment
	Category	Documentation
	Severity	Yellow
	Code Smell	Missing Documentation

18	Error Message	Method ‘System.err.println‘ should not be called
	Occurrence	20
	Type	Discouraged APIs
	Category	Correctness
	Severity	Yellow
	Code Smell	Standard Output/Error Usage

19	Error Message	Violation of nesting depth threshold of 3: 4
	Occurrence	15
	Type	Metric Violations/Nesting Depth
	Category	Structure
	Severity	Yellow
	Code Smell	Excessive Nesting

20	Error Message	Violation of method length threshold (source lines of code) of x:y
	Occurrence	10
	Type	Metric Violations/LSL
	Category	Structure
	Severity	Yellow
	Code Smell	Long Method

21	Error Message	Star import of ‘java.awt.*‘ should not be used
	Occurrence	11
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Wildcard Import

22	Error Message	Star import of ‘java.awt.event.*‘ should not be used
	Occurrence	11
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Wildcard Import

23	Error Message	Star import of ‘javax.swing.*‘ should not be used
	Occurrence	11
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Wildcard Import

24	Error Message	Star import of ‘org.payroll.*‘ should not be used
	Occurrence	9
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Wildcard Import

25	Error Message	Star import of ‘java.util.*‘ should not be used
	Occurrence	6
	Type	Bad Practice
	Category	Comprehensibility
	Severity	Yellow
	Code Smell	Wildcard Import

4 Re-engineering Methods for Undesirables

We demonstrated our findings in this area and related them to their reengineering rule, reengineering rule type, rule tag, undesirable severity, unfavourable likelihood, and the name of the team member who refactored it. For the most part, this information was obtained via evaluating the following forum https://rules.sonarsource.com/java and mapping the discovery to the best-fitting reengineering rule.

1	Findings	Empty block: method
	Re-engineering Rule Description	For a number of reasons, a method may not have a method body. Unintentional mistakes ought to be fixed, as this one is. It is not now and never will be supported. The override method is intentionally empty. In this situation, a nested comment must give a justification for the blank.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Suspicious
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mahavir Patel

2	Findings	Static variable dbManager is not initialized
	Re-engineering Rule Description	Ensure that all static variables, including dbManager, are initialized before they are accessed or used. Proper initialization prevents potential runtime errors and guarantees that static variables have valid values at all times.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Major
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mahavir Patel

3	Findings	Attribute ‘btn delete‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mahavir Patel

4	Findings	Attribute ‘btn exit‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mahavir Patel

5	Findings	Non-constant public attribute dbManager
	Re-engineering Rule Description	Class attributes, such as dbManager, should be declared as private or protected, and access to them should be provided through getter and setter methods. Avoid exposing class attributes directly as public to ensure proper encapsulation and maintain data integrity.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Major
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mahavir Patel

6	Findings	Attribute ‘btn OK‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mahavir Patel

7	Findings	Attribute ‘btn update‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

8	Findings	Method System.exit should not be called
	Re-engineering Rule Description	There are several implementation classes available for interfaces to meet different needs. Specified performance aspects are guaranteed by some implementations, whereas specified behaviours, such immutability, are guaranteed by others. Therefore, before invoking a "optional" method, a developer should make sure that the implementation class on which the call is made implements it.
	Re-engineering Rule Type	Bug
	Rule Tag	N/A
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

9	Findings	Attribute 'lbl basic salary' violates naming convention.*'
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

10	Findings	Attribute 'lbl da' violates naming convention.*
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

11	Findings	Attribute ‘lbl dep name‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

12	Findings	Attribute ‘lbl department‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Preet Angad Singh Nanda

13	Findings	Attribute ‘lbl em‘ violates naming convention.*‘
	Re-engineering Rule Description	Teams may work together effectively using shared coding norms. This rule determines if every constant name satisfies the given regular expression.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Convention
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Varun Pandey

14	Findings	Clone with 2 instances of length x
	Re-engineering Rule Description	It should not be changed to ”clone”
	Re-engineering Rule Type	Code Smell
	Rule Tag	Suspicious
	Undesirable Severity	Yellow
	Undesirable Likelihood	High
	Refactored By	Varun Pandey

15	Findings	else statement without braces
	Re-engineering Rule Description	It can lead to confusion and unintended consequences, especially when developers later modify the code by adding more statements to the conditional branch. Code must consistently enclose the body of an else statement with braces, even if it contains only a single statement. This ensures code readability, avoids potential bugs due to indentation issues, and future-proofs the code against unintended side effects when modifying the conditional branch.
	Re-engineering Rule Type	Code Quality
	Rule Tag	Minor
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Varun Pandey

16	Findings	if statement without braces
	Re-engineering Rule Description	It can lead to confusion and unintended consequences, especially when developers later modify the code by adding more statements to the conditional branch. Code must consistently enclose the body of an if statement with braces, even if it contains only a single statement. This ensures code readability, avoids potential bugs due to indentation issues, and future-proofs the code against unintended side effects when modifying the conditional branch.
	Re-engineering Rule Type	Code Quality
	Rule Tag	Minor
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Varun Pandey

17	Findings	Interface comment missing
	Re-engineering Rule Description	There is no JavaDoc accessible for the models, interfaces, or classes.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Unused
	Undesirable Severity	Yellow
	Undesirable Likelihood	High
	Refactored By	Varun Pandey

18	Findings	Method ‘System.err.println‘ should not be called
	Re-engineering Rule Description	There are several implementation classes available for interfaces to meet different needs. Specified performance aspects are guaranteed by some implementations, whereas specified behaviours, such immutability, are guaranteed by others. Therefore, before invoking a ”optional” method, a developer should make sure that the implementation class on which the call is made implements it.
	Re-engineering Rule Type	Bug
	Rule Tag	N/A
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Varun Pandey

19	Findings	Violation of nesting depth threshold of 3: 4
	Re-engineering Rule Description	Excessive nesting can lead to code that is harder to read, understand, and maintain. It can also increase the chances of introducing bugs and make the code more challenging to modify or refactor in the future.
	Re-engineering Rule Type	Bugs
	Rule Tag	Checkstyle
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mir Pasad

20	Findings	Violation of method length threshold (source lines of code) of x:y
	Re-engineering Rule Description	a particular method in the codebase has exceeded the allowed or recommended length limit set by the coding standards or guidelines of the project. It may also be an indicator that the method is doing too much and should be refactored into smaller, more focused methods with clear responsibilities.
	Re-engineering Rule Type	Code Smell
	Rule Tag	Checkstyle
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mir Pasad

21	Findings	Star import of ‘java.awt.*‘ should not be used
	Re-engineering Rule Description	Similar to database queries, using wildcards to import software libraries or packages may be a sign of laziness on the part of individuals who don’t want to take the effort to import a specific component (such as import java.awt.*) to be utilised in their program. Too many wildcard imports might result in problems when your programs are being compiled.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Slow Compilation
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mir Pasad

22	Findings	Star import of ‘java.awt.event.*‘ should not be used
	Re-engineering Rule Description	Similar to database queries, using wildcards to import software libraries or packages may be a sign of laziness on the part of individuals who don’t want to take the effort to import a specific component (such as import java.awt.Event.*) to be utilised in their program. Too many wildcard imports might result in problems when your programs are being compiled.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Slow Compilation
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mir Pasad

23	Findings	Star import of ‘javax.swing.*‘ should not be used
	Re-engineering Rule Description	Similar to database queries, using wildcards to import software libraries or packages may be a sign of laziness on the part of individuals who don’t want to take the effort to import a specific component (such as import java.swing.*) to be utilised in their program. Too many wildcard imports might result in problems when your programs are being compiled.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Slow Compilation
	Undesirable Severity	Yellow
	Undesirable Likelihood	Medium
	Refactored By	Mir Pasad

24	Findings	Star import of ‘org.payroll.*‘ should not be used
	Re-engineering Rule Description	Java programs compile-time speed could be somewhat impacted by the use of wildcard imports, but runtime performance won’t be impacted at all. When it comes to clean coding, you have to be careful how you employ it. .
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Slow Compilation
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mir Pasad

25	Findings	Star import of ‘java.util.*‘ should not be used
	Re-engineering Rule Description	Java programs compile-time speed could be somewhat impacted by the use of wildcard imports, but runtime performance won’t be impacted at all. When it comes to clean coding, you have to be careful how you employ it.
	Re-engineering Rule Type	Vulnerability
	Rule Tag	Slow Compilation
	Undesirable Severity	Yellow
	Undesirable Likelihood	Low
	Refactored By	Mir Pasad

5 Location of Source code Undesirables

The table below summarises all of the undesirable sites. The location of the undesirable is indicated by saying the folder name, then the file where it is found within this folder, and where this undesirable is found, as well as how many lines of source code it contains.

No.	Location	File Name	Lines of Code	Source Lines of Code	Number of Findings
1.	/sources/org/payroll/	DatabaseManager.java	287	254	53
2.	/sources/org/payroll/	LoginFrame.java	94	80	17
3.	/sources/org/payroll/	Main.java	14	8	5
4.	/sources/org/payroll/	MainFrame.java	249	180	24
5.	/sources/org/payroll/departments/	DeleteDepartmentFrame.java	65	54	14
6.	/sources/org/payroll/departments/	ModifyDepartmentFrame.java	114	101	32
7.	/sources/org/payroll/departments/	NewDepartmentFrame.java	110	98	32
8.	/sources/org/payroll/employees/	DeleteEmployeeFrame.java	87	75	18
9.	/sources/org/payroll/employees/	NewEmployeeFrame.java	88	76	22
10.	/sources/org/payroll/employees/	UpdateEmployeeFrame.java	111	98	30
11.	/sources/org/payroll/preferences/	ChangePasswordFrame.java	147	130	34
12.	/sources/org/payroll/preferences/	DeleteLoginIdFrame.java	81	71	18
13.	/sources/org/payroll/preferences/	NewLoginIdFrame.java	173	151	37
Total			1620	1376	336

6 Software Metric log

A software meter is a quantitative or countable measure of program properties. Software metrics are useful for a variety of purposes, including monitoring software performance, planning work items, measuring productivity, and many more. Candidate R (Employee Payroll system) was subjected to TeamScale (A software metric analysis tool) to determine variances in system quality before and after refactoring. The representation below highlights the relevant changes in the candidate R's software metrics.

6.1 Before Refactoring

Path	Files	Lines of Code	Source Lines of Code	Longest Method Length	Maximum Nesting Depth	Change Count	Number of Findings	Findings Density
Summary	13	1.6k	1.4k	74	6	0	336	205.9
departments	3	292	253	37	4	0	78	267.1
employees	3	289	249	37	4	0	70	242.2
preferences	3	404	352	51	6	0	89	220.3
DatabaseManager.java	1	288	254	34	2	0	53	184
LoginFrame.java	1	95	80	14	3	0	17	178.9
Main.java	1	15	8	2	0	0	5	333.3
MainFrame.java	1	249	180	74	2	0	24	96.4

Figure 2: Package Metrics of **Candidate R - Employee Payroll System** before Refactoring

Path	Files	Last Change Date	Number of Findings	Number of Findings (Red)	Number of Findings (Yellow)	Findings Density	Findings Density (Red)	Findings Density (Yellow)
Summary	13	Jul 26 2023 23:22	336	2	334	205.9	1.2	204.7
departments	3	Jul 26 2023 23:22	78	0	78	267.1	0	267.1
employees	3	Jul 26 2023 23:22	70	0	70	242.2	0	242.2
preferences	3	Jul 26 2023 23:22	89	2	87	220.3	5	215.3
DatabaseManager.java	1	Jul 26 2023 23:22	53	0	53	184	0	184
LoginFrame.java	1	Jul 26 2023 23:22	17	0	17	178.9	0	178.9
Main.java	1	Jul 26 2023 23:22	5	0	5	333.3	0	333.3
MainFrame.java	1	Jul 26 2023 23:22	24	0	24	96.4	0	96.4

Figure 3: Quality Metrics of **Candidate R - Employee Payroll System** before Refactoring

6.2 After Refactoring

Path	Files	Lines of Code	Source Lines of Code	Longest Method Length	Maximum Nesting Depth	Change Count	Number of Findings	Findings Density
Summary	13	1.9k	1.4k	71	6	65	54	28.8
departments	3	345	258	29	4	15	18	52.2
employees	3	337	247	27	4	13	14	41.5
preferences	3	443	351	37	6	15	17	38.4
DatabaseManager.java	1	351	247	29	2	10	0	0
LoginFrame.java	1	110	84	14	3	3	1	9.1
Main.java	1	19	8	2	0	1	1	52.6
MainFrame.java	1	268	191	71	2	8	3	11.2

Figure 4: Package Metrics of **Candidate R - Employee Payroll System** after Refactoring

Path	Files	Number of Findings	Number of Findings (Red)	Number of Findings (Yellow)	Findings Density	Findings Density (Red)	Findings Density (Yellow)
Summary	13	54	2	52	28.8	1.1	27.8
departments	3	18	0	18	52.2	0	52.2
employees	3	14	0	14	41.5	0	41.5
preferences	3	17	2	15	38.4	4.5	33.9
DatabaseManager.java	1	0	0	0	0	0	0
LoginFrame.java	1	1	0	1	9.1	0	9.1
Main.java	1	1	0	1	52.6	0	52.6
MainFrame.java	1	3	0	3	11.2	0	11.2

Figure 5: Quality Metrics of **Candidate R - Employee Payroll System** after Refactoring

7 Refactoring report

We can see the left-hand side of this final report, which has the data we initially got when we ran the system via team scale, and the right-hand side, which contains the data after refactoring. The number of lines of code has increased because we had to provide functionality for Java documentation.

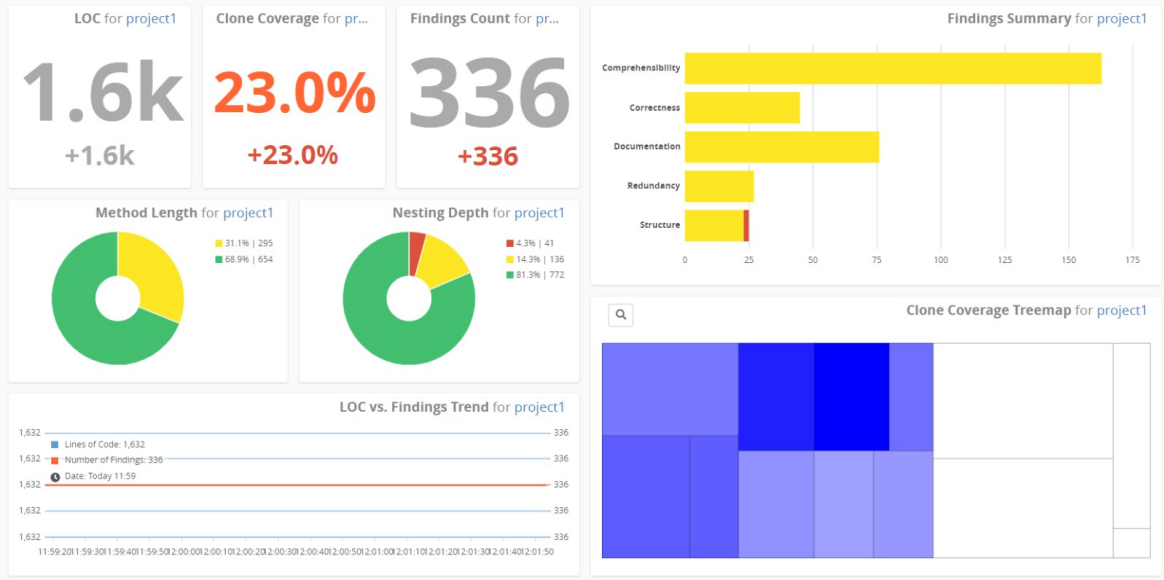


Figure 6: Before Refactoring

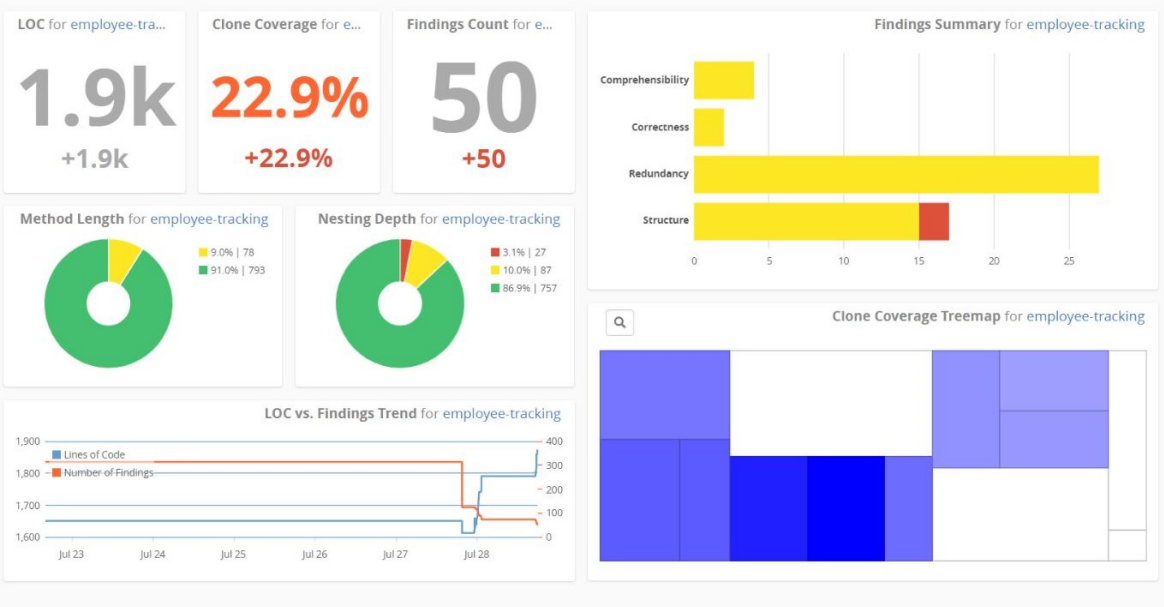
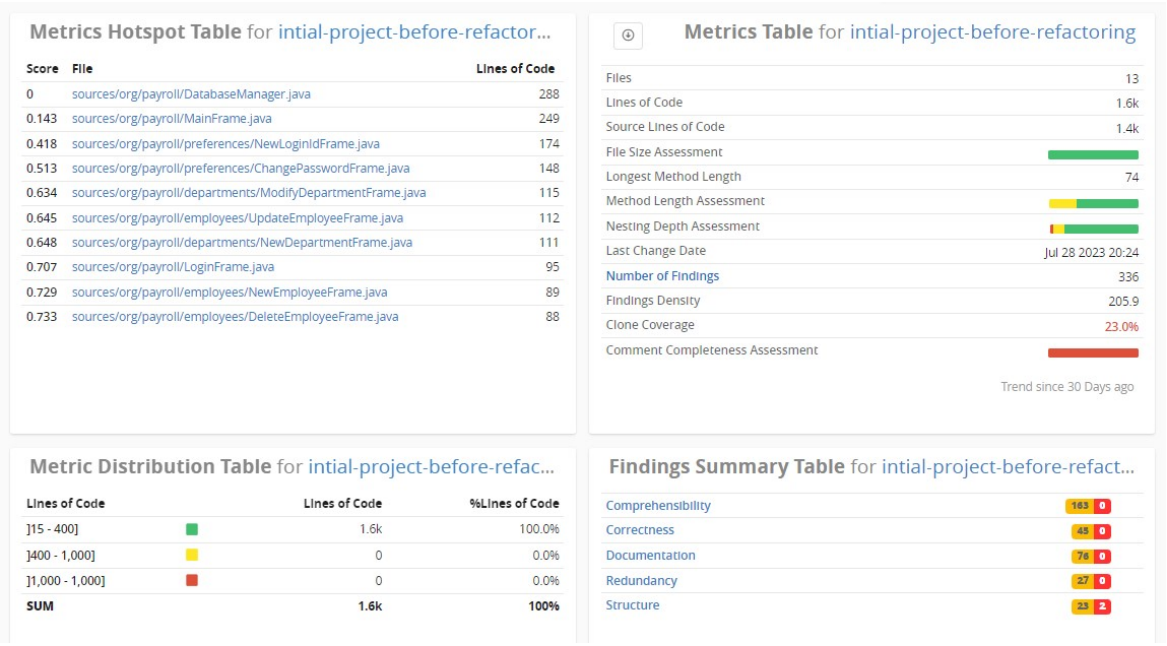


Figure 7: After Refactoring



Metrics Hotspot Table for intial-project-before-refactor...

Score	File	Lines of Code
0	sources/org/payroll/DatabaseManager.java	288
0.143	sources/org/payroll/MainFrame.java	249
0.418	sources/org/payroll/preferences/NewLoginIdFrame.java	174
0.513	sources/org/payroll/preferences/ChangePasswordFrame.java	148
0.634	sources/org/payroll/departments/ModifyDepartmentFrame.java	115
0.645	sources/org/payroll/employees/UpdateEmployeeFrame.java	112
0.648	sources/org/payroll/departments/NewDepartmentFrame.java	111
0.707	sources/org/payroll/LoginFrame.java	95
0.729	sources/org/payroll/employees/NewEmployeeFrame.java	89
0.733	sources/org/payroll/employees/DeleteEmployeeFrame.java	88

Metrics Table for intial-project-before-refactoring

Files	13
Lines of Code	1.6k
Source Lines of Code	1.4k
File Size Assessment	
Longest Method Length	74
Method Length Assessment	
Nesting Depth Assessment	
Last Change Date	Jul 28 2023 20:24
Number of Findings	336
Findings Density	205.9
Clone Coverage	23.0%
Comment Completeness Assessment	

Trend since 30 Days ago

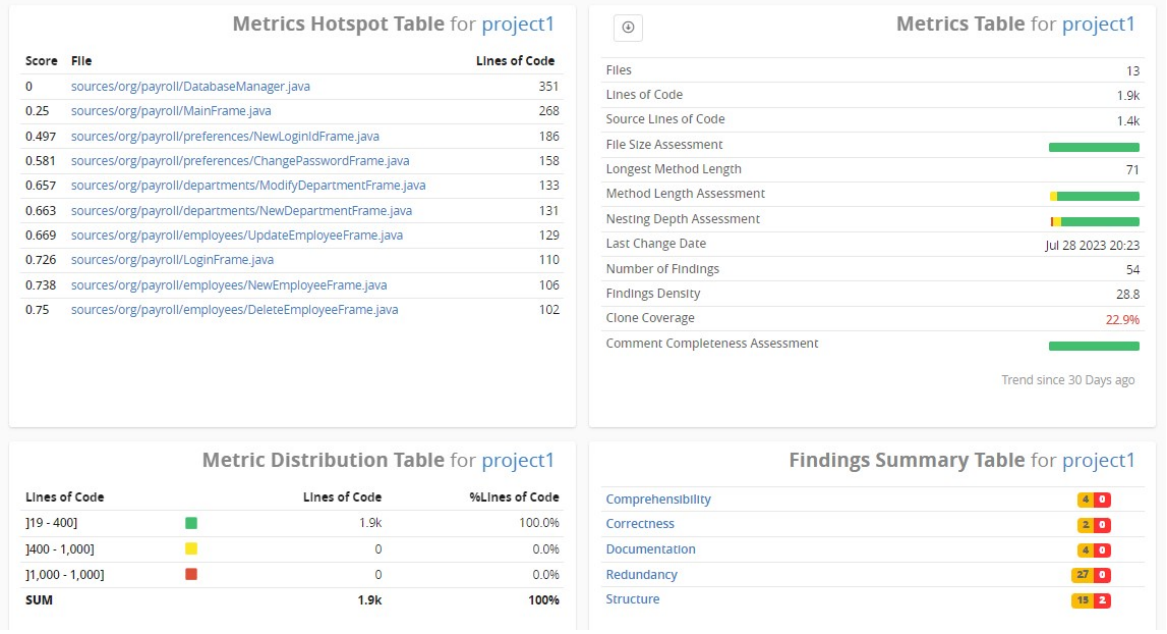
Metric Distribution Table for intial-project-before-refac...

Lines of Code		Lines of Code	%Lines of Code
]15 - 400]		1.6k	100.0%
]400 - 1,000]		0	0.0%
]1,000 - 1,000]		0	0.0%
SUM		**1.6k**	**100%**

Findings Summary Table for intial-project-before-refact...

Comprehensibility	163	0
Correctness	45	0
Documentation	76	0
Redundancy	27	0
Structure	23	2

Figure 8: Software Metrics Before Refactoring



Metrics Hotspot Table for project1

Score	File	Lines of Code
0	sources/org/payroll/DatabaseManager.java	351
0.25	sources/org/payroll/MainFrame.java	268
0.497	sources/org/payroll/preferences/NewLoginIdFrame.java	186
0.581	sources/org/payroll/preferences/ChangePasswordFrame.java	158
0.657	sources/org/payroll/departments/ModifyDepartmentFrame.java	133
0.663	sources/org/payroll/departments/NewDepartmentFrame.java	131
0.669	sources/org/payroll/employees/UpdateEmployeeFrame.java	129
0.726	sources/org/payroll/LoginFrame.java	110
0.738	sources/org/payroll/employees/NewEmployeeFrame.java	106
0.75	sources/org/payroll/employees/DeleteEmployeeFrame.java	102

Metrics Table for project1

Files	13
Lines of Code	1.9k
Source Lines of Code	1.4k
File Size Assessment	
Longest Method Length	71
Method Length Assessment	
Nesting Depth Assessment	
Last Change Date	Jul 28 2023 20:23
Number of Findings	54
Findings Density	28.8
Clone Coverage	22.9%
Comment Completeness Assessment	

Trend since 30 Days ago

Metric Distribution Table for project1

Lines of Code		Lines of Code	%Lines of Code
]19 - 400]		1.9k	100.0%
]400 - 1,000]		0	0.0%
]1,000 - 1,000]		0	0.0%
SUM		**1.9k**	**100%**

Findings Summary Table for project1

Comprehensibility	4	0
Correctness	2	0
Documentation	4	0
Redundancy	27	0
Structure	15	2

Figure 9: Software Metrics After Refactoring

8 Software Specifications

8.1 Tools Used to Refactor the Candidate R

1. Eclipse IDE: Eclipse is a computer programming integrated development environment. It includes a base workspace as well as other plugins that allow the system to be customised. The debuggers major feature assisted us in numerous ways in refining the code. Its fantastic user interface makes it simple for developers to debug, track, and browse through various files.

2. TeamScale: It is a software intelligence platform that provides transparency into code quality and the underlying software development process. This allows developers, testers, and managers to better understand and manage technical debt in their systems. It is the engine for incremental analysis. Because it is directly tied to the version control system, it assesses each commit progressively. This enables TeamScale to provide timely feedback and highlight the fundamental causes of emergent problems or deteriorating trends on a commit-based basis.

3. Sonar Lint Integration Eclipse: Sonar Lint is a free and open-source IDE extension that detects and fixes quality and security issues as you code. Sonar Lint, like a spell checker, squiggles defects and delivers real-time feedback and explicit remedial assistance to deliver clean code from the start. Nowadays, code quality is an essential component of any development pipeline. It's about preventing defects from affecting end users, preventing security risks from leaking into the wild, and making your code easier to maintain. Static Code Analysis is critical in this case.This is where Sonar Lint comes in help.

4. Overleaf: Overleaf is a cloud-based collaborative LaTeX editor for drafting, revising, and publishing scientific publications. It collaborates with a variety of scientific publishers to provide official journal LaTeX templates as well as direct submission links.Overleaf was created by John Hammersley and John Lees-Miller, who began working on it as Write LaTeX in 2011.Write LaTeX Limited is a firm. We utilised it to document our work in Latex, and the biggest advantage is that we can share it with each team member and work simultaneously. Furthermore, the editing packages are already installed in overleaf.

5. GIT: Git is a free and open-source distributed version control system that can handle everything from tiny to extremely large projects quickly and efficiently. Git is simple to use, has a small footprint, and delivers lightning-fast speed. It outperforms SCM tools such as Subversion, CVS, and Git. Perforce and ClearCase, for example, offer low-cost local branching, convenient staging areas, and numerous processes.

8.2 Software Quality Standard to Refactor Candidate R

Figure 10: ISO/IEC 25010:2011-Software Quality Requirements and Evaluation

1. A standard for software quality is ISO 25010, "Systems and software engineering - Systems and software Quality Requirements and Evaluation - System and software quality models." It outlines the models, including characteristics and sub-characteristics for both software product quality and software quality in use, and offers helpful guidance on how to use the quality models. The ISO 25010 describes two quality models:

2. The five characteristics that make up the quality in use model, some of which are further subdivided. An eight-trait (each of which is further subdivided into sub-characteristics) model of product quality.

3. Eight product quality criteria and 31 sub-characteristics make up ISO 25010: Functionality, dependability, performance effectiveness, usability, security, compatibility, maintenance requirements, and portability.

4. According to ISO guidelines, the candidate R after refactoring must to possess the following characteristics: The phrase "maintenanceability" refers to how quickly a product or system may be modified to improve, correct, or adapt to changes imposed by the environment and users. We have provided a handbook that is easy for the next developer to understand, as well as a solution that is modular and easily reused.

9 Refactored source code of R

Candidates' R source code before and after the refactoring is included in the section below, and a GitHub repository link is supplied below each of them.

Source Code of Candidate R before Refactoring
Link : https://github.com/Selected-Source-Code-Repositories/Employee-Payroll-System

Source Code of Candidate R After Refactoring
Link : https://github.com/Candidate-System-Employee-Management-System

10 Conclusion

In conclusion, the software re-engineering effort aimed at improving an employee payroll system created in Java was a success. The team's comfort with the project and the code's adherence to project specifications influenced the decision to choose this system. The purpose of the re-engineering effort was to make the code more maintainable and future-proof.

The team used two code analysis tools, Teamscale and SonarLint, to detect and address flaws in the code. They chose Teamscale after comparing the results since it provided more extensive information about problems, severity levels, locations, and viable repairs, as well as better visualisations for the issues.

During the code auditing process, several undesirable errors were detected, including empty method blocks, uninitialized static variables, and naming convention violations. To address these challenges efficiently, the team used re-engineering principles and standards, with different team members taking responsibility for reworking individual problems.

The system's maintainability and functionality have improved as a result of the re-engineering initiatives. The codebase is now compliant with coding standards, and several code smells have been addressed. Furthermore, the team eliminated redundant and error-prone code practises, improving the system's overall quality.

In summary, the employee payroll system has been successfully re-engineered to improve maintainability, stability, and usability by extensive analysis, diligent reworking, and adherence to code standards. The accomplishments of the project have ensured the system's durability and increased its performance for future usage.

11 References

1. Understanding Maintainability by Dr. Pankaj Kamthan [2023]
2. ISO Software Quality Standard
   https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
3. TeamScale Documentation
   https://docs.teamscale.com/
4. Improving Code Quality Using Teamscale
   https://docs.teamscale.com/introduction/improving-software-quality/
5. Teamscale Integration for Eclipse
   https://docs.teamscale.com/getting-started/eclipse/
6. Integrating Sonar to Eclipse
   https://subscription.packtpub.com/book/programming/integrating-sonar-to-eclipse
7. Catching Issues in the IDE with SonarLint
   https://docs.sonarcloud.io/improving/sonarlint/
8. Everything you need to know about Code Smells
   https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/
9. ISO/IEC 25010 Software Quality Model
   https://blog.codacy.com/iso-25010-software-quality-model/
10. Manage Team's Project on Trello
    https://trello.com/soen-6431