Concordia University
Engineering and
Computer Science

SOEN 6431 : SOFTWARE COMPREHENSION AND MAINTENANCE

Summer 2022

Deliverable - 2 : Reengineering Operationalization

Github Link

Authors

Manimaran Palani
Iphigenia Pappas
Heet Patel
Kevinkumar Patel
Venis Patel

https://www.overleaf.com/project/610304de4e6b8d24f7c781b6

Contents

1. Introduction

Software Re-engineering plays a very important role in improving the maintainability of a software system. It gives the software system a new life through alterations thus contributing to the maintainability side of the software. We chose Online banking system as candidate R for our project(more information available in Deliverable-1).

The Online Banking System is a banking portal on the web which manages the customer profiles and their respective transactions. It is highly scalable and secured with the help of Spring Security. The main features of this project includes validation of login form, viewing customer profile, viewing transaction details of the customer ,viewing balance of the customer, approval of the changes in the personal information by the customer. The core objective of this project is to maintain a personal account in the bank. The system also provides access to the customer to create an account, deposit/withdrawal of cash from the account, along with the luxury to view reports of all the accounts available.

The reason we chose the online banking system as our candidate system is not only because it was one of the better choices from the software provided by the team (rejected reasons can be found below for all other systems) but for several other important reasons. To name a few, this system met all the requirements, was complex enough and for us to have a lot to work on the core reengineering process, without being too much where we would get lost in the code. There is not too much spaghetti code, although it may be optimized to less lines of code. This project was between 1000 to 2000 lines of code which was the target desire for our system. We also were easily able to locate the 25 undesirables to fix for this system. Each team member was able to identify 5 distinct undesirables that we went on to fix later. Lastly, this system was written, in large majority, in our programming language of choice - Java. The architecture of this system also contains multiple distinct aspects that we can easily categorize in two main groups: managing customer profiles and managing transactions. This system also contains a certain level of security as it maintains credential information by validating a login form. In summary, this system met all the requirements and more and was well structured enough for us to clearly identify our undesirables and work on them.

2. Source Code Undesirables Summary

Our team used Team Scale in order to identify all undesirables we would then fix for this assignment. Below, is the list of different findings that were corrected. These findings varied from more severe undesirables, such as lack of security, to things as simple as commented out blocks of that could not have been there. The software allowed us to identify them all, along with the number of occurrences.

In this section, we have also included the type of the undesirable, the category, code smell type which are all more or less indicating the reason that this would be considered an undesirable. We have also included a summary of each Code smell in order to give a little more detail about the undesirable. We had a total of 21 different types of findings that include all of this information. We have also included some graphs which allows us to visualize the undesirables better.

1	**Findings**	Clone with 2 instances of length 16
	Occurences	4
	Type	Redundancy/Clones
	Category	Code Duplication
	Code smell type	***Test smells - Test Code Duplication***
	Code smell summary	Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem.

2	**Findings**	Empty block : method
	Occurences	2
	Type	Code Anomalies/cqse-empty-block
	Category	Comprehensibility
	Code smell type	***Configuration Smells - Unnecessary Abstraction***
	Code smell summary	A class, ’define’, or module must contain declarations or statements specifying the properties of a desired system. An empty class, ’define’, or module shows the presence of unnecessary abstraction smell and thus must be removed.

3	**Findings**	Commented out code
	Occurences	2
	Type	Comments/Commented out code
	Category	Comprehensibility
	Code smell type	***Implementation Smells - Comments***
	Code smell summary	This smell occurs when comments are used as deodorant to explain the bad code.

4	**Findings**	Star import of ‘javax.persistence.*‘ should not be used
	Occurences	2
	Type	Code Anomalies/cqse-no-star-imports
	Category	Bad Practice
	Code smell type	***Design Smells - Obsolete imports***
	Code smell summary	This smell occurs when certain classes are no longer used in a software system but loaded due to improper signature of import statements. Classes that are no longer in use will burden the system with obviously obsolete functionality.

5	**Findings**	Unused import: ‘com.userfront.domain.SavingsTransaction‘
	Occurences	1
	Type	Code Anomalies/cqse-java-unused-imports
	Category	Bad Practice
	Code smell type	***Architecture Smells - unused packages***
	Code smell summary	Packages that are not in use burden the system with clearly obsolete functionality.

6	**Findings**	Method ‘System.out.println‘ should not be called
	Occurences	3
	Type	Code Anomalies/cqse-java-unwanted-method-calls
	Category	Discouraged APIs
	Code smell type	***Design Smells - Poltergeist***
	Code smell summary	Irrelevant classes: An irrelevant class is a class that does not have any meaningful behaviour in the design. These types of classes are characterised for being composed only of get, set and/or print methods

7	**Findings**	‘SimpleDateFormat‘ constructor should specify ‘Locale‘
	Occurences	1
	Type	Code Anomalies/cqse-avoid-creating-simple-date-format-without-locale
	Category	API misuse
	Code smell type	***Design Smells -Divergent Change***
	Code smell summary	You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type, you have to change the methods for finding, displaying, and ordering products.

8	**Findings**	Interface comment missing
	Occurences	151
	Type	Comments/Missing interface comments
	Category	Comment completeness
	Code smell type	***Shortage smells - Debt***
	Code smell summary	Similarly, to Chant that covers up imperfect fragments with comments in natural language, there could be pieces missing entirely from the grammar and replaced with comments. If the comments admit clearly what is missing, use searchable tags like “TODO” or “FIXME” and are intended to use as a backlog.

9	**Findings**	TODO Auto-generated method stub
	Occurences	3
	Type	Comments/Task tags
	Category	Dosumentation
	Code smell type	***Implementation smells - Comments***
	Code smell summary	This smell occurs when comments are used as deodorant to explain the bad code.

10	**Findings**	Catch clause catches generic exception ‘Exception‘
	Occurences	1
	Type	Code Anomalies/cqse-catch-high-level-exception
	Category	Imprecise Handling
	Code smell type	***Implementation smells - Catch block***
	Code smell summary	This smell occurs when a catch block of an exception is improperly handled with non-matching or non-precise exception libraries.

11	**Findings**	Logger should be specified with ‘UserServiceImpl.class‘
	Occurences	1
	Type	Code Anomalies/cqse-logger-with-wrong-specified-class
	Category	Logging
	Code smell type	***Implementation smells - Attribute name and type are opposite***
	Code smell summary	The name of an attribute is in contradiction with its type as they contain antonyms. \Example: attribute start of type Association End. The use of antonyms can induce wrong assumptions.

12	**Findings**	Clone with 2 instances of length 28
	Occurences	2
	Type	Redundancy/Clones
	Category	Code Duplication
	Code smell type	***Test smells - Test code duplication***
	Code smell summary	Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem.

13	**Findings**	Private field ‘env‘ is never read.
	Occurences	1
	Type	Code Anomalies/cqse-java-avoid-unused-private-fields
	Category	Unused code
	Code smell type	***Design smells - unutilized abstraction***
	Code smell summary	This smell arises when a variable is left unused (either not directly used or not reachable).

14	**Findings**	Empty block : Constructor
	Occurences	4
	Type	Code Anomalies/cqse-empty-block
	Category	Comprehensibility
	Code smell type	***Configuration smells - Unnecessary abstraction***
	Code smell summary	A class, ’define’, or module must contain declarations or statements specifying the properties of a desired system. An empty class, ’define’, or module shows the presence of unnecessary abstraction smell and thus must be removed.

15	**Findings**	Unused import: ‘com.userfront.domain.PrimaryTransaction‘
	Occurences	1
	Type	Code Anomalies/cqse-java-unused-imports
	Category	Unused Code
	Code smell type	***Architecture smells - Unused Packages***
	Code smell summary	Packages that are not in use burden the system with clearly obsolete functionality.

16	**Findings**	Name ‘com.userfront.service.UserServiceImpl‘ violates naming convention. Should be one of ‘[a-z][a-z_0-9.]*‘
	Occurences	5
	Type	Naming/JAVA
	Category	Comprehensibility
	Code smell type	***Configuration Smells (Implementation) - Inconsistent Naming Convention***
	Code smell summary	The used naming convention deviates from the recommended naming convention.

17	**Findings**	Remove this use of ‘getBytes‘
	Occurences	1
	Type	Compatability issues
	Category	Correctness
	Code smell type	***Implementation Smells - “Get” - more than an accessor***
	Code smell summary	A getter that performs actions other than returning the corresponding attribute without documenting it.Example: method getImageData which, no matter the attribute value, every time returns a new object.

18	**Findings**	Use ‘BigDecimal.ZERO‘ instead of creating new object
	Occurences	2
	Type	Code Anomalies/cqse-unnecessary-big-integer-instantiation
	Category	Performance
	Code smell type	***Performance Smells - Unnecessary Processing***
	Code smell summary	This smell occurs when processing is not needed or not needed at that time. Solution: Delete the extra processing steps, reorder steps to detect unnecessary steps earlier, or restructure to delegate those steps to a background task.

19	**Findings**	Comment should not contain ‘/*‘
	Occurences	1
	Type	Code Anomalies/cqse-nested-comment
	Category	Malformed Comments
	Code smell type	***Configuration Smells (Implementation) - Improper Quote Usage***
	Code smell summary	Single and double quotes are not used properly. For example, Boolean values should not be quoted, and variable names should not be used in single quoted strings.

20	**Findings**	The ‘printStackTrace()‘ method should not be called
	Occurences	1
	Type	Code Anomalies/cqse-print-stack-trace
	Category	Imprecise Handling
	Code smell type	***Implementation Smells - Incomplete Library Class***
	Code smell summary	The author of the library has not provided the features you need or has refused to implement them. invoking printStackTrace() changes the destination pointed to by System.err by redirecting the process to a file/device whose contents may be ignored by personnel.

21	**Findings**	Throw of generic exception “Exception”
	Occurences	1
	Type	Code Anomalies/cqse-throw-high-level-java-exceptions
	Category	Imprecise Handling
	Code smell type	***Implementation Smells - Catch Block***
	Code smell summary	This smell occurs when a catch block of an exception is improperly handled with non-matching or non-precise exception libraries.

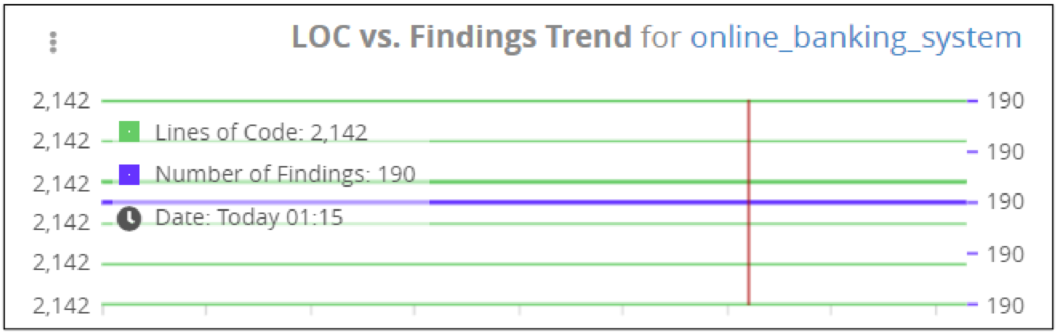


Figure 1: Lines of Code vs. Findings Trend of **Candidate R** - Online Banking System

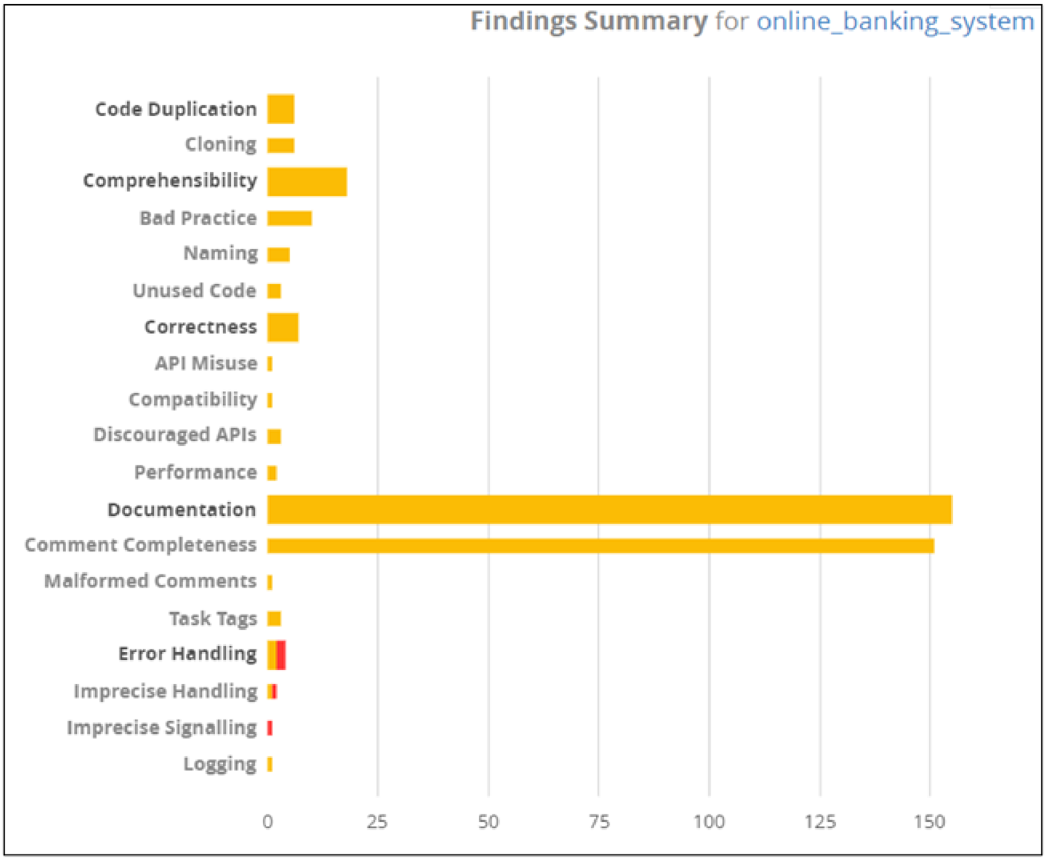
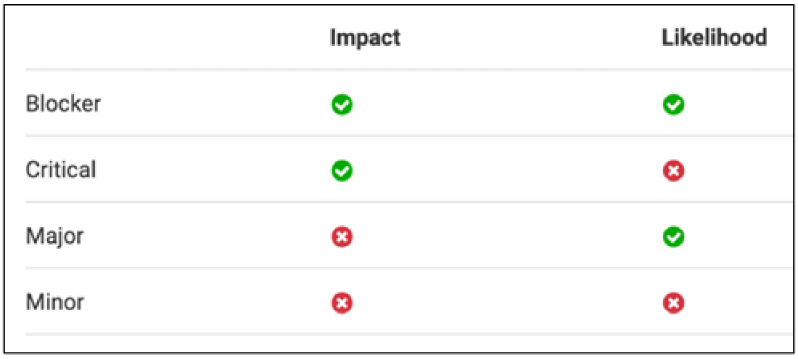


Figure 2: Findings Summary of **Candidate R** - Online Banking System

3. Re-engineering Methods for Undesirables

In this section, we have displayed our findings and matched them to their **reengineering rule, reengineering rule type, rule tag, undesirable severity, undesirable likelihood**, and we have also included the name of the team member that refactored it. For the most part, this information was retrieved by analysing the following forum `https://rules.sonarsource.com/java` and mapping the finding to the reengineering rule that fits the best description.

As our system was entirely in java, we used Java static code analysis (Unique rules to find bugs, Vulnerabilities, Security Hotspots, and code Smells in your JAVA code). We based ourselves on the following graph in order to properly identify the value. This Graph was retrieved in the following forum`https://docs.sonarqube.org/latest/user-guide/rules/`



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

Figure 3: Severity category by Sonar Qube

1	**Findings**	Clone with 2 instances of length 16
	Reengineering Rule Description	”clone” should not be overridden
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Suspicious
	Undesirable Severity	Blocker
	Undesirable Likelihood	High
	Refactored By	Manimaran Palani

2	**Findings**	Clone with 2 instances of length 28
	Reengineering Rule Description	”clone” should not be overridden
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Suspicious
	Undesirable Severity	Blocker
	Undesirable Likelihood	High
	Refactored By	Manimaran Palani

3	**Findings**	Empty block: method
	Reengineering Rule Description	There are several reasons for a method not to have a method body. It is an unintentional omission and should be fixed to prevent an unexpected behaviour in production. It is not yet, or never will be, supported. In this case an UnsupportedOperationException should be thrown. The method is an intentionally blank override. In this case a nested comment should explain the reason for the blank override.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Suspicious
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Manimaran Palani

4	**Findings**	Private field ‘env‘ is never read.
	Reengineering Rule Description	A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only to then overwrite it or throw it away, could indicate a serious error in the code. Even if it’s not an error, it is at best a waste of resources. Therefore, all calculated values should be used.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Swe, Cert , Unused
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Manimaran Palani

5	**Findings**	Commented Out code
	Reengineering Rule Description	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.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Unused
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Manimaran Palani

6	**Findings**	Empty block : Constructor
	Reengineering Rule Description	There are several reasons for a method not to have a method body: It is an unintentional omission and should be fixed to prevent an unexpected behaviour in production. It is not yet, or never will be, supported. In this case an UnsupportedOperationException should be thrown. The method is an intentionally blank override. In this case a nested comment should explain the reason for the blank override.
	Reengineering Ruule Type	***Code smell***
	Rule Tag	Suspicious
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Iphigenia Pappas

7	**Findings**	Star import of ‘javax.persistence.*‘ should not be used
	Reengineering Rule Description	On one side, Spring MVC automatically bind request parameters to beans declared as arguments of methods annotated with @RequestMapping. Because of this automatic binding feature, it’s possible to feed some unexpected fields on the arguments of the @RequestMapping annotated methods. For this reason, using @Entity or @Document objects as arguments of methods annotated with @RequestMapping should be avoided.
	Reengineering Rule Type	***Vulnerability***
	Rule Tag	cwe, spring, OWASP
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Iphigenia Pappas

8	**Findings**	Unused import: ‘com.userfront.domain.PrimaryTransaction‘
	Reengineering Rule Description	Unused parameters are misleading. Whatever the values passed to such parameters, the behaviour will be the same.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Cert unused
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Iphigenia Pappas

9	**Findings**	Unused import: ‘com.userfront.domain.SavingsTransaction‘
	Reengineering Rule Description	Unused parameters are misleading. Whatever the values passed to such parameters, the behaviour will be the same.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Cert unused
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Iphigenia Pappas

10	**Findings**	Name ‘com.userfront.service.UserServiceImpl‘ violates naming convention. Should be one of ‘[a-z][a-z_0-9.]*‘
	Reengineering Rule Description	Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression.
	Reengineering Rule Type	***Code Smell***
	Rule Tag	Convention
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Heet Patel

11	**Findings**	Method ‘System.out.println‘ should not be called
	Reengineering Rule Description	The Java Collections framework defines interfaces such as java.util.List or java.util.Map. Several implementation classes are provided for each of those interfaces to fill different needs: some of the implementations guarantee a few given performance characteristics, some others ensure a given behaviour, for example immutability. When calling one of the ”optional” methods, a developer should therefore make sure that the implementation class on which the call is made indeed supports this method.
	Reengineering Rule Type	***Bug***
	Rule Tag	N/A
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Heet Patel

12	**Findings**	Remove this use of ‘getBytes‘
	Reengineering Rule Description	Getters and setters provide a way to enforce encapsulation by providing public methods that give controlled access to private fields. However, in classes with multiple fields it is not unusual that copy and paste is used to quickly create the needed getters and setters, which can result in the wrong field being accessed by a getter or setter. This rule raises an issue in any of these cases: A setter does not update the field with the corresponding name. A getter does not access the field with the corresponding name.
	Reengineering Rule Type	***Bug***
	Rule Tag	Pitfall
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Heet Patel

13	**Findings**	‘SimpleDateFormat‘ constructor should specify ‘Locale‘
	Reengineering Rule Description	Non-abstract classes and enums with non-static, private members should explicitly initialize those members, either in a constructor or with a default value.
	Reengineering Rule Type	***Code smell***
	Rule Tag	Pitfall
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Heet Patel

14	**Findings**	Use ‘BigDecimal.ZERO‘ instead of creating new object
	Reengineering Rule Description	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.
	Reengineering Rule Type	***Code smell***
	Rule Tag	Design
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Kevinkumar Patel

15	**Findings**	Interface comment missing
	Reengineering Rule Description	JavaDoc is not available for the classes, interface, and models
	Reengineering Rule Type	***Code smell***
	Rule Tag	unused
	Undesirable Severity	Minor
	Undesirable Likelihood	High
	Refactored By	Kevinkumar Patel

16	**Findings**	Comment should not contain ‘/*‘
	Reengineering Rule Description	Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression.
	Reengineering Rule Type	***Code smell***
	Rule Tag	Convention
	Undesirable Severity	Critical
	Undesirable Likelihood	Rare
	Refactored By	Kevinkumar Patel

17	**Findings**	TODO Auto-generated method stub
	Reengineering Rule Description	TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed.
	Reengineering Rule Type	***Code smell***
	Rule Tag	swe
	Undesirable Severity	Minor
	Undesirable Likelihood	Rare
	Refactored By	Kevinkumar Patel

18	**Findings**	Catch clause catches generic exception ‘Exception‘
	Reengineering Rule Description	Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors.
	Reengineering Rule Type	***Code smell***
	Rule Tag	swe, error handling-cert
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Venis Patel

19	**Findings**	Throw of generic exception Exception
	Reengineering Rule Description	Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors.
	Reengineering Rule Type	***Code smell***
	Rule Tag	swe, error handling-cert
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Venis Patel

20	**Findings**	Logger should be specified with ‘UserServiceImpl.class‘
	Reengineering Rule Description	It is convention to name each class's logger for the class itself. Doing so allows you to set up clear, communicative logger configuration. Naming loggers by some other convention confuses configuration and using the same class name for multiple class loggers prevents the granular configuration of each class' logger. Some libraries, such as SLF4J warn about this, but not all do. This rule raises an issue when a logger is not named for its enclosing class.
	Reengineering Rule Type	***Code smell***
	Rule Tag	Confusing
	Undesirable Severity	Minor
	Undesirable Likelihood	Rare
	Refactored By	Venis Patel

21	**Findings**	The ‘printStackTrace()‘ method should not be called
	Reengineering Rule Description	Because printf-style format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that result in the wrong strings being created. This rule statically validates the correlation of printf-style format strings to their arguments when calling the format(...) methods of java.util.Formatter, java.lang.String, java.io.PrintStream, MessageFormat, and java.io.PrintWriter classes and the printf(...) methods of java.io.PrintStream or java.io.PrintWriter classes.
	Reengineering Rule Type	***Code smell***
	Rule Tag	cert, confusing
	Undesirable Severity	Major
	Undesirable Likelihood	High
	Refactored By	Venis Patel

4. Location of Source code Undesirables

The table below summarizes the locations of all the undesirable. The place where the undesirable is located is specified by mentioning the folder name, then the file where it is found within this folder and mentions where this undesirable is found and how many lines of source code it includes.

As you can see at the bottom of the table, there were a **total of 2142 lines of code** worth of undesirables and **190 different findings**. This huge number of undesirables was brought down to 8 undesirables after our maintenance and corrections. The reason we have left the 8 out of 190 there, was simply because changing these 8 would alter the functionality of this system which was out of scope of what was asked in the project description.

No.	Folder	File	Lines of Code	Source Lines of code	Number of Findings
1	userfront/config	RequestFilter.java	51	40	9
2	userfront/config	SecurityConfig.java	77	58	8
3	userfront/controller	AccountController.java	90	67	7
4	userfront/controller	AppointmentController.java	56	40	4
5	userfront/controller	HomeController.java	85	63	6
6	userfront/controller	TransferController.java	125	92	9
7	userfront/controller	UserController.java	50	34	3
8	userfront/dao	AppointmentDao.java	13	7	2
9	userfront/dao	PrimaryAccountDao.java	13	6	1
10	userfront/dao	PrimaryTransactionDao.java	13	7	2
11	userfront/dao	RecipientDao.java	16	9	4
12	userfront/dao	RoleDao.java	10	6	2
13	userfront/dao	SavingsAccountDao.java	13	6	1
14	userfront/dao	SavingsTransactionDao.java	14	7	2
15	userfront/dao	UserDao.java	14	9	4
16	userfront/domain/ security	Authority.java	19	12	2
17	userfront/domain/ security	Role.java	50	32	4
18	userfront/domain/ security	UserRole.java	59	39	5
19	userfront/domain	Appointment.java	86	67	1
20	userfront/domain	PrimaryAccount.java	66	46	2
21	userfront/domain	PrimaryTransaction.java	108	82	5
22	userfront/domain	Recipient.java	86	65	1
23	userfront/domain	SavingsAccount.java	63	46	2
24	userfront/domain	SavingsTransaction.java	106	82	5
25	userfront/domain	User.java	208	158	4

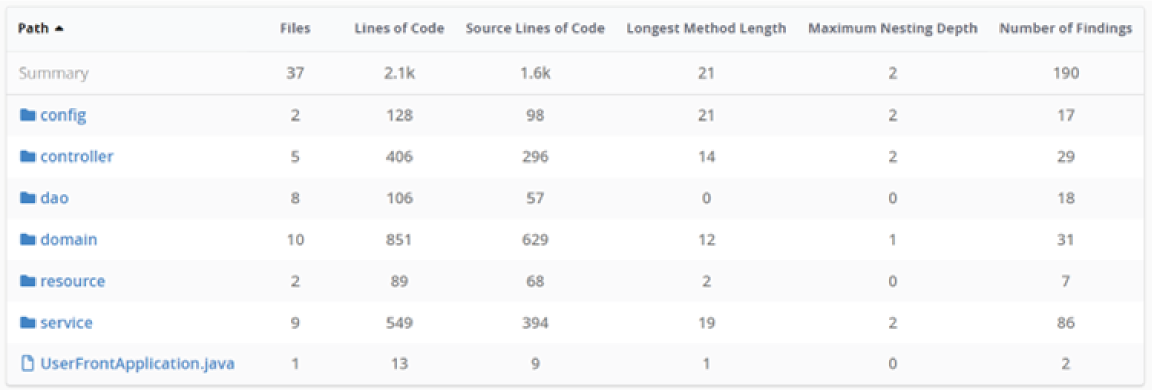
No.	Folder	File	Lines of Code	Source Lines of code	Number of Findings
26	userfront/resource	AppointmentResource.java	34	25	3
27	userfront/resource	UserResource.java	55	43	4
28	userfront/service/ UserServiceImpl	AccountServiceImpl.java	112	81	10
29	userfront/service/ UserServiceImpl	AppointmentServiceImpl.java	36	26	6
30	userfront/service/ UserServiceImpl	TransactionServiceImpl.java	145	110	15
31	userfront/service/ UserServiceImpl	UserSecurityService.java	33	25	2
32	userfront/service/ UserServiceImpl	UserServiceImpl.java	121	91	16
33	userfront/service/ UserServiceImpl	AccountService.java	18	12	7
34	userfront/service	AppointmentService.java	16	9	5
35	userfront/service	TransactionService.java	36	22	13
36	userfront/service	UserService.java	32	18	12
37	userfront	UserFrontApplication.java	13	9	2
		Total lines of code	**2142**	***Total Findings***	**190**

5. Software Metric log

A software metric is a measure of software characteristics that are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

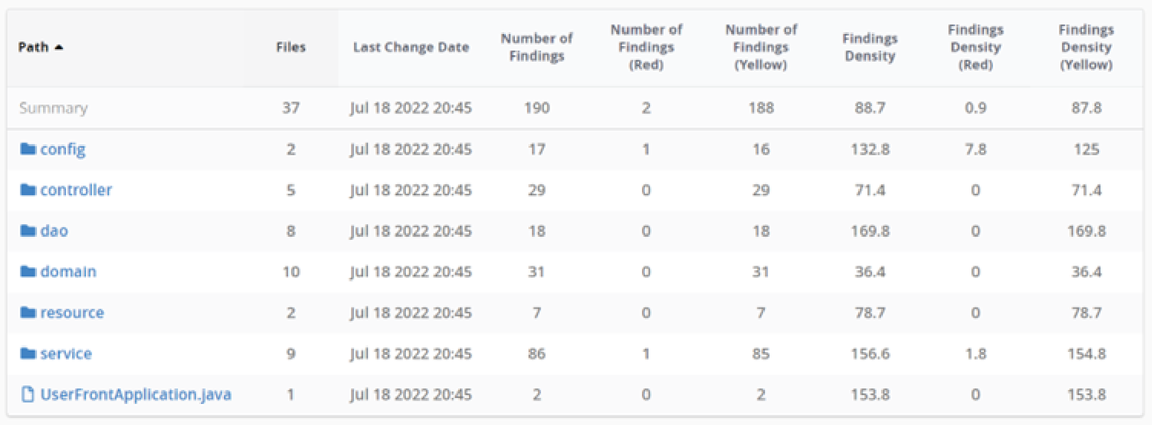
With respect to candidate R, (online banking system)it has been subjected to TeamScale (A software metric analysing tool) to find out the deviations in the quality of the system before and after refactoring. The below representation emphasizes the relevant changes in the software metrics of the candidate R.

Before Refactoring



Path ▲	Files	Lines of Code	Source Lines of Code	Longest Method Length	Maximum Nesting Depth	Number of Findings
Summary	37	2.1k	1.6k	21	2	190
config	2	128	98	21	2	17
controller	5	406	296	14	2	29
dao	8	106	57	0	0	18
domain	10	851	629	12	1	31
resource	2	89	68	2	0	7
service	9	549	394	19	2	86
UserFrontApplication.java	1	13	9	1	0	2

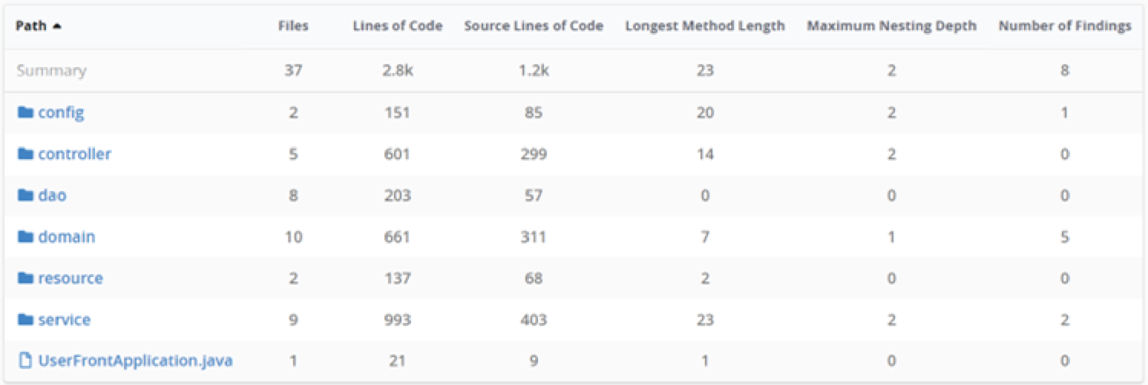
Figure 4: Package Metrics of **Candidate R** - Online Banking System before Refactoring



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	37	Jul 18 2022 20:45	190	2	188	88.7	0.9	87.8
config	2	Jul 18 2022 20:45	17	1	16	132.8	7.8	125
controller	5	Jul 18 2022 20:45	29	0	29	71.4	0	71.4
dao	8	Jul 18 2022 20:45	18	0	18	169.8	0	169.8
domain	10	Jul 18 2022 20:45	31	0	31	36.4	0	36.4
resource	2	Jul 18 2022 20:45	7	0	7	78.7	0	78.7
service	9	Jul 18 2022 20:45	86	1	85	156.6	1.8	154.8
UserFrontApplication.java	1	Jul 18 2022 20:45	2	0	2	153.8	0	153.8

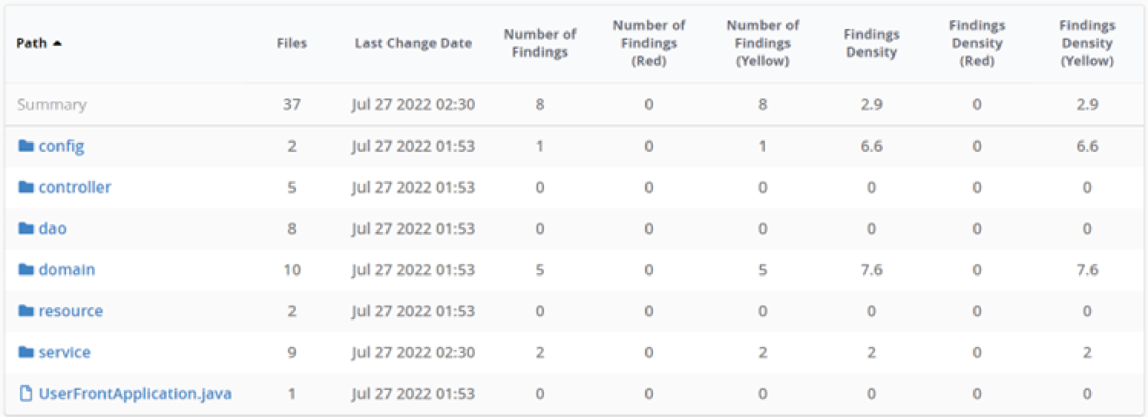
Figure 5: Quality Metrics of **Candidate R** - Online Banking System before Refactoring

After Refactoring



Path ▲	Files	Lines of Code	Source Lines of Code	Longest Method Length	Maximum Nesting Depth	Number of Findings
Summary	37	2.8k	1.2k	23	2	8
config	2	151	85	20	2	1
controller	5	601	299	14	2	0
dao	8	203	57	0	0	0
domain	10	661	311	7	1	5
resource	2	137	68	2	0	0
service	9	993	403	23	2	2
UserFrontApplication.java	1	21	9	1	0	0

Figure 6: Package Metrics of **Candidate R** - Online Banking System after 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	37	Jul 27 2022 02:30	8	0	8	2.9	0	2.9
config	2	Jul 27 2022 01:53	1	0	1	6.6	0	6.6
controller	5	Jul 27 2022 01:53	0	0	0	0	0	0
dao	8	Jul 27 2022 01:53	0	0	0	0	0	0
domain	10	Jul 27 2022 01:53	5	0	5	7.6	0	7.6
resource	2	Jul 27 2022 01:53	0	0	0	0	0	0
service	9	Jul 27 2022 02:30	2	0	2	2	0	2
UserFrontApplication.java	1	Jul 27 2022 01:53	0	0	0	0	0	0

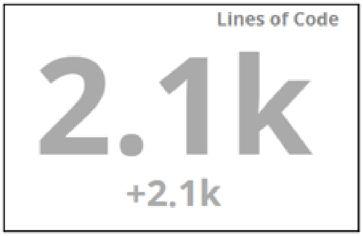
Figure 7: Quality Metrics of **Candidate R** - Online Banking System after Refactoring

6. Refactoring report

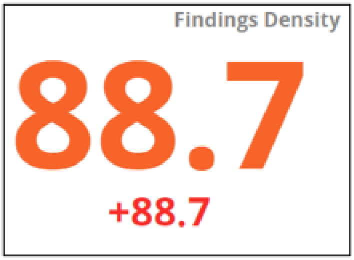
In this final report we can see the left-hand side which holds the data we retrieved when running the system through team scale initially, and the right-hand side includes after refactoring. The lines of code have gone up, as we had to add functionalities with regard to API security and Java documentation. 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.

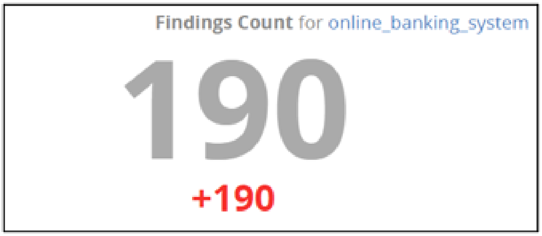
Before Refactoring



Lines of Code
2.1k
+2.1k

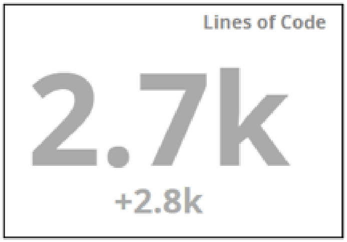


Findings Density
88.7
+88.7

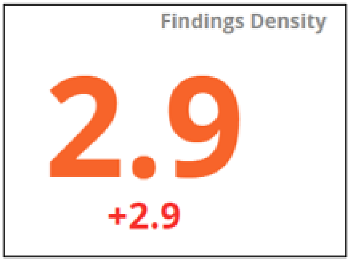


Findings Count for online_banking_system
190
+190

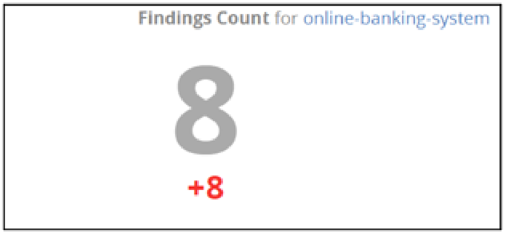
After Refactoring



Lines of Code
2.7k
+2.8k

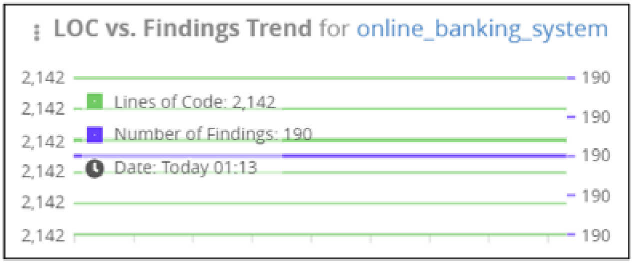


Findings Density
2.9
+2.9



Findings Count for online-banking-system
8
+8

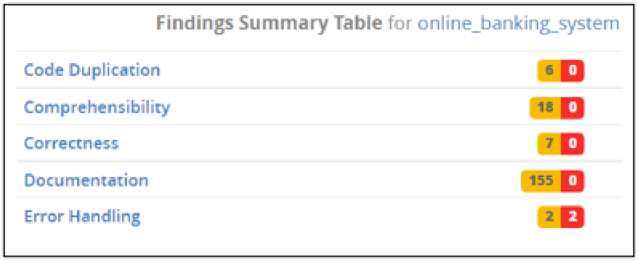
Before Refactoring



🟩 Lines of Code: 2,142

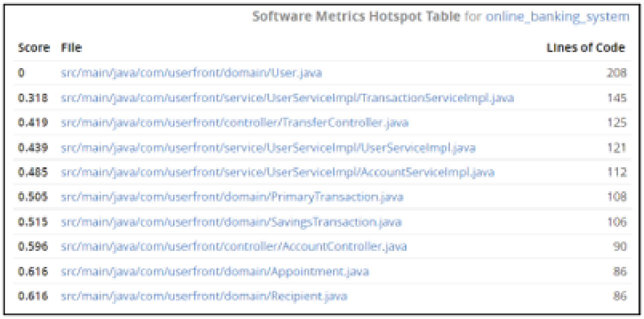
🟦 Number of Findings: 190

Date: Today 01:13



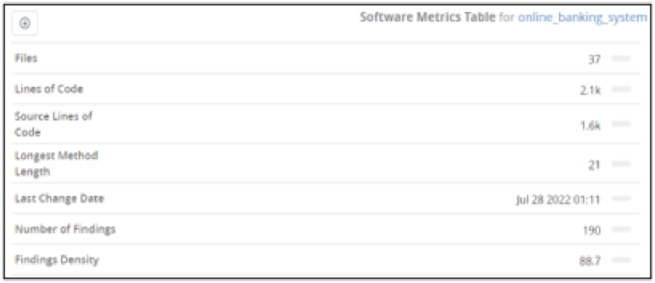
Findings Summary Table for online_banking_system

Code Duplication	6	0
Comprehensibility	18	0
Correctness	7	0
Documentation	155	0
Error Handling	2	2



Software Metrics Hotspot Table for online_banking_system

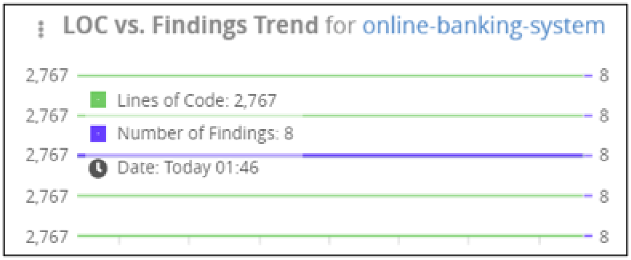
Score	File	Lines of Code
0	src/main/java/com/userfront/domain/User.java	208
0.318	src/main/java/com/userfront/service/UserServiceImpl/TransactionServiceImpl.java	145
0.419	src/main/java/com/userfront/controller/TransferController.java	125
0.439	src/main/java/com/userfront/service/UserServiceImpl/UserServiceImpl.java	121
0.485	src/main/java/com/userfront/service/UserServiceImpl/AccountServiceImpl.java	112
0.505	src/main/java/com/userfront/domain/PrimaryTransaction.java	108
0.515	src/main/java/com/userfront/domain/SavingsTransaction.java	106
0.596	src/main/java/com/userfront/controller/AccountController.java	90
0.616	src/main/java/com/userfront/domain/Appointment.java	86
0.616	src/main/java/com/userfront/domain/Recipient.java	86



Software Metrics Table for online_banking_system

Files	37
Lines of Code	2.1k
Source Lines of Code	1.6k
Longest Method Length	21
Last Change Date	Jul 28 2022 01:11
Number of Findings	190
Findings Density	88.7

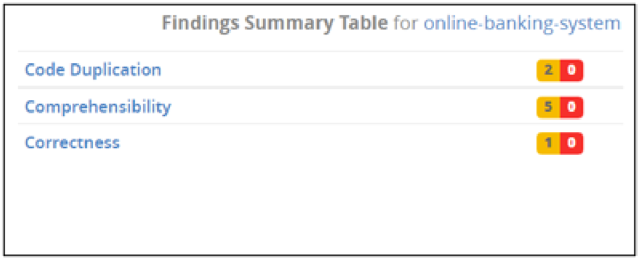
After Refactoring



🟩 Lines of Code: 2,767

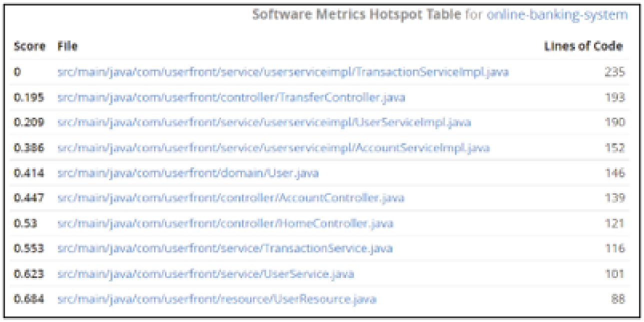
🟦 Number of Findings: 8

Date: Today 01:46



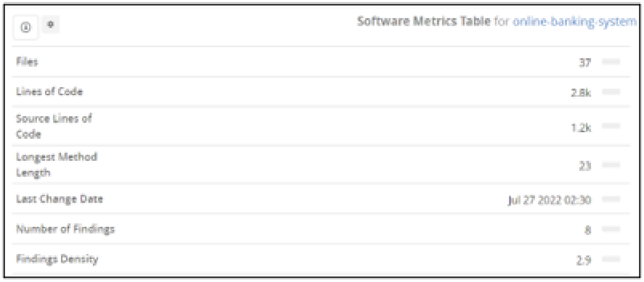
Findings Summary Table for online-banking-system

Code Duplication	2	0
Comprehensibility	5	0
Correctness	1	0



Software Metrics Hotspot Table for online-banking-system

Score	File	Lines of Code
0	src/main/java/com/userfront/service/userserviceimpl/TransactionServiceImpl.java	235
0.195	src/main/java/com/userfront/controller/TransferController.java	193
0.209	src/main/java/com/userfront/service/userserviceimpl/UserServiceImpl.java	190
0.386	src/main/java/com/userfront/service/userserviceimpl/AccountServiceImpl.java	152
0.414	src/main/java/com/userfront/domain/User.java	146
0.447	src/main/java/com/userfront/controller/AccountController.java	139
0.53	src/main/java/com/userfront/controller/HomeController.java	121
0.553	src/main/java/com/userfront/service/TransactionService.java	116
0.623	src/main/java/com/userfront/service/UserService.java	101
0.684	src/main/java/com/userfront/resource/UserResource.java	88



Software Metrics Table for online-banking-system

Files	37
Lines of Code	2.8k
Source Lines of Code	1.2k
Longest Method Length	23
Last Change Date	Jul 27 2022 02:30
Number of Findings	8
Findings Density	2.9

7. Software specifications

7.1 Tools Used to Refactor the Candidate R

1. Eclispse IDE : Eclipse is an integrated development environment used in computer programming. It contains a base workspace and so many plugins for the system to get a customized environment. The main feature of the debugger helped us in improving the code in many ways. The wonderful user interface of it makes easy for the developer to debug, track, and navigate through different files.

2. Teamscale : Teamscale is a software intelligence platform, that is, it creates transparency on code quality and the underlying software development process. This makes it possible for developers, testers, and managers to better understand and control technical debt of their systems. It is the incremental analysis engine. It is directly connected to the version control system and, hence, analyses each commit incrementally. This enables Teamscale to provide rapid feedback and reveal the root causes on commit-based for emerging problems or deteriorating trends.

3. Sonar Lint Integration in Eclipse : Sonar Lint is a Free and Open-Source IDE extension that identifies and helps you fix quality and security issues as you code. Like a spell checker, Sonar Lint squiggles flaws and provides real-time feedback and clear remediation guidance to deliver clean code from the get-go. Code Quality is an integral part of any software pipeline nowadays. It's about preventing bugs from impacting end users, preventing security vulnerabilities from making it to the open world, and also easing the maintainability of your code. Static Code Analysis plays an essential role here.This is where Sonar Lint is a very handy tool.

4. Checkstyle : The check style development tool is a plugin added into the IDE. It is able to check the coding standard automatically. It makes the software developer work's easier by indicating the design problems, such code formats,layout,etc.It helps with auditing code structure in classes and methods and also diminishes the chances of formatting problem to occur.

5. Overleaf : Overleaf is a collaborative cloud-based LaTeX editor used for writing, editing, and publishing scientific documents. It partners with a wide range of scientific publishers to provide official journal LaTeX templates, and direct submission links .Overleaf was conceived by John Hammersley and John Lees-Miller, who started developing it in 2011as Write LaTeX, through their company Write LaTeX Limited. We used it to document our work in Latex , plus the main benefit of this is that we can share it across each team member, and everyone can work simultaneously. Also, the packages need for editing are already installed in overleaf.

6. GIT : Git is a free and open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Git is easy to learn and has a tiny footprint with lightning-fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

7.2 ISO/IEC 25010:2011 Software Quality Standard to Refactor Candidate R

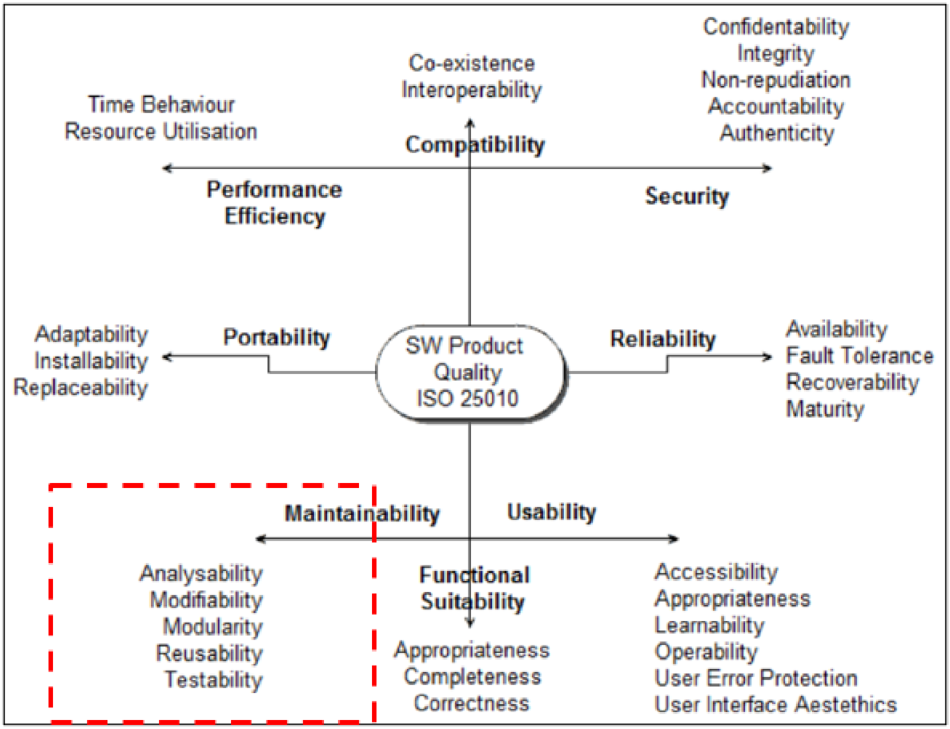


Figure 10: **ISO/IEC 25010:2011**-Software Quality Requirements and Evaluation **(SQuaRE)**

ISO 25010, titled "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models", is a software quality standard. It describes the models, consisting of characteristics and sub-characteristics, for both software product quality, and software quality in use together with practical guidance on the use of the quality models. ISO25010 describes two quality models:

1. The quality in use model composed of five characteristics (some of which are further sub-divided into sub-characteristics).

2. A product quality model composed of eight characteristics (which are further sub-divided into sub-characteristics).

ISO 25010 is made up of eight product quality characteristics and 31 sub-characteristics:

1. Functional Suitability
2. Reliability
3. Performance Efficiency
4. Usability

5. Security
6. Compatibility
7. **Maintainability**
8. Portability

The characteristics which the candidate R has to possess after refactoring according to the ISO standards is:

Maintainability refers to how well a product or system can be modified to improve, correct, or adapt to changes in the environment as well as requirements. Our system is modular, reusable easily, the coding is understandable and also, we have added the document that can be understood by the next developer.

8. Refactored source code of R

In the below section, candidate's R source code below refactoring and after refactoring has been attached.

Source Code of Candidate R after Refactoring
Source Code of Candidate R before Refactoring

9. References

1. D. Korolev, "How do you define code quality?," 09 02 2014. [Online]..
2. ISO/IEC 25010:2011," 03 2011. [Online]. .
3. ISO/IEC 9126-1:2001," 06 2001. [Online].
XML Tutorial," 2017. [Online]..
4. What is the optimal size of a software development team," 16 05 2009. [Online]..
5. R. . Norvig, The intelligent agent paradigm, 2003, pp. 27, 32–58, 968–972 .
6. "Teamscale Documentation" [Online]..
7. "Installing-the-plug-in" [Online].
8. "Connecting TeamScale to Eclipse" [Online].
9. "Sonar Installation" [Online].
10. "Connecting Sonar to Eclipse Issues" [Online].
11. "Configuration of Checkstyle in Eclipse" [Online].
12. "GIT Documentation" [Online].
13. "ISO 25010:2011" [Online].