
# SOEN 6431 : SOFTWARE COMPREHENSION AND MAINTENANCE

**Summer 2023**

## Deliverable - 2 : Reengineering Operationalization

Github Link

***Authors***

Unnati Chaturvedi
Poornaa Himadri Bhattacharya
Jatin Chhabra
Vikram Singh Brahm

https://www.overleaf.com/project/64bd84f86776aa25fb29b116

# Contents

# 1. Introduction

This report presents the findings and outcomes of the reengineering project focused on enhancing the source code of the Hotel Management System. The primary objective of this project was to improve the maintainability and overall quality of the system's source code, following the ISO/IEC 25010 Standard guidelines. To achieve this goal, the project team employed various reengineering techniques, with a strong emphasis on refactoring.

Throughout the project, effective project management practices were employed to ensure smooth progress. Regular team meetings were conducted at least once a week, enabling effective communication and tracking of the Déjà Vu Hotel Management System's reengineering process.

The project's key aspects included the identification of code smells and undesirables in the Hotel Management System's source code, both through manual inspection and automated code analysis tools. The team diligently applied appropriate refactoring methods to eliminate these code smells and improve the codebase's overall quality. After each modification, thorough debugging, reviewing, and testing were conducted to ensure the system's stability and functionality.

This report includes an organized list of detected code smells and undesirables in the Hotel Management System's source code. Additionally, it provides a comprehensive list of reengineering methods employed, along with their respective impact on the codebase's maintainability and efficiency. The project's successful completion highlights the importance of maintaining clean, efficient, and maintainable source code in software systems. The methodologies implemented and the experience gained during this project can serve as valuable insights for future software engineering endeavors.

# 2. Methodology for Hotel Management System Report

Methodology followed for the report is given as follows:

- Research and Analysis
  - Conducted a detailed analysis of the existing Hotel Management System codebase to identify potential areas of improvement.
  - Researched best practices and coding standards for Java programming to ensure adherence to industry norms.
  - Explored various code quality assessment tools and selected DesigniteJava for detecting code smells and potential maintainability issues.
- Refactoring Planning
  - Developed a comprehensive plan for refactoring the Hotel Management System code, outlining the specific areas to be addressed and the refactoring techniques to be applied.
  - Prioritized code smells based on their impact on maintainability and efficiency, focusing on critical issues first.

- Refactoring Implementation
  - Utilized IntelliJ IDEA 2023 as the Integrated Development Environment (IDE) for the refactoring process.
  - Applied various refactoring techniques to eliminate code smells, improve code structure, and enhance maintainability.
  - Ensured that each refactoring step was carefully reviewed and tested to avoid introducing new issues.
- Testing
  - Created tests to verify if everything is working fine after the validation.
- Code Review
  - Collaborated as a team to conduct a thorough code review, identifying areas of improvement and providing constructive feedback.
  - Reviewed the naming conventions, code organization, error handling mechanisms, input validation, and overall code quality.
- Quality Metrics Analysis
  - Utilized DesigniteJava to collect software metrics before and after refactoring.
  - Analyzed the metrics to evaluate the impact of the refactoring process on code quality, maintainability, and performance.
- Documentation
  - Documented the entire refactoring process, including the refactoring techniques applied and the rationale behind each change.
  - Created a comprehensive report summarizing the findings, improvements, and the final state of the Hotel Management System code.
- Final Review and Delivery
  - Conducted a final review of the refactored code to ensure that all identified issues were addressed.
  - Delivered the final refactored version of the Hotel Management System code with enhanced maintainability, efficiency, and quality.

The methodology followed a systematic approach, with a strong emphasis on collaboration, testing, and code quality assessment, resulting in a significantly improved Hotel Management System codebase.

## 3. Working of the candidate system

The **hotelManagement** package in the candidate system has the following classes:

- **Main Class (Main.java):**
  - The `Main` class is the entry point of the application and contains the main method.
  - It initializes various data structures and objects, such as ArrayLists for customers, hotel rooms, bookings, and previous bookings.
  - The program starts with a main loop that presents a menu to the user and prompts for their choice.
  - The user can either choose options related to hotel employees or exit the program.
- **HotelLogic Class (HotelLogic.java):**
  - The `HotelLogic` class handles the business logic for managing customers, hotel rooms, and bookings.
  - It includes methods to add, remove, edit, and view customer information.
  - Methods for adding, removing, and editing room details are also present.
  - The class manages booking-related operations, such as making a new booking, removing a booking, checking in, and checking out.
- **Customer Class (Customer.java):**
  - The `Customer` class represents a customer of the hotel and stores attributes such as account number, name, contact details, etc.
- **Room Class (Room.java):**
  - The `Room` class represents a hotel room and stores attributes like room number, number of beds, balcony availability, price per night, and booking status.
- **Booking Class (Booking.java):**
  - The `Booking` class represents a booking made by a customer for a particular room on a specific date.
  - It stores attributes like account number, booked date, room number, and check-in status.
- **ReadFile Class (ReadFile.java) and CreateFile Class (CreateFile.java):**
  - These classes handle file I/O for reading and writing bookings information to a file named "allBookings.txt."
- **PrintMenus Class (PrintMenus.java):**
  - The PrintMenus class contains methods to print various menus, providing options for hotel employees to choose from.

## 4. Location of the source code undesirables before refactoring

Table 1: Code Smells in `hotelManagement` before refactoring

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 1 | Long Statement | hotelManagement | CreateFile | addRecord |
| 2 | Long Parameter List | hotelManagement | Customer | Customer |
| 3 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 4 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 5 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 6 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 7 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 8 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 9 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 10 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 11 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 12 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 13 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 14 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 15 | Complex Method | hotelManagement | HotelLogic | addCustomer |
| 16 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 17 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 18 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 19 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 20 | Long Statement | hotelManagement | HotelLogic | addRoom |
| 21 | Complex Method | hotelManagement | HotelLogic | makeBooking |
| 22 | Complex Method | hotelManagement | HotelLogic | viewInfoAboutCustomer |
| 23 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 24 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 25 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 26 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 27 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 28 | Complex Method | hotelManagement | HotelLogic | editCustomInfo |
| 29 | Long Statement | hotelManagement | HotelLogic | editCustomInfo |
| 30 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 31 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 32 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 33 | Complex Method | hotelManagement | HotelLogic | editBookingInfo |
| 34 | Long Identifier | hotelManagement | HotelLogic | editBookingInfo |
| 35 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 36 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 37 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 38 | Magic Number | hotelManagement | Main | main |
| 39 | Magic Number | hotelManagement | Main | main |
| 40 | Magic Number | hotelManagement | Main | main |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 41 | Magic Number | hotelManagement | Main | main |
| 42 | Magic Number | hotelManagement | Main | main |
| 43 | Magic Number | hotelManagement | Main | main |
| 44 | Magic Number | hotelManagement | Main | main |
| 45 | Magic Number | hotelManagement | Main | main |
| 46 | Magic Number | hotelManagement | Main | main |
| 47 | Magic Number | hotelManagement | Main | main |
| 48 | Magic Number | hotelManagement | Main | main |
| 49 | Magic Number | hotelManagement | Main | main |
| 50 | Magic Number | hotelManagement | Main | main |
| 51 | Magic Number | hotelManagement | Main | main |
| 52 | Long Parameter List | hotelManagement | Room | Room |
| 53 | Long Parameter List | hotelManagement | Room | Room |
| 54 | Long Statement | hotelManagement | CreateFile | addRecord |
| 55 | Long Parameter List | hotelManagement | Customer | Customer |
| 56 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 57 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 58 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 59 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 60 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 61 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 62 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 63 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 64 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 65 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 66 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 67 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 68 | Complex Method | hotelManagement | HotelLogic | addCustomer |
| 69 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 70 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 71 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 72 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 73 | Long Statement | hotelManagement | HotelLogic | addRoom |
| 74 | Complex Method | hotelManagement | HotelLogic | makeBooking |
| 75 | Complex Method | hotelManagement | HotelLogic | viewInfoAboutCustomer |
| 76 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 77 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 78 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 79 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 80 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 81 | Complex Method | hotelManagement | HotelLogic | editCustomInfo |
| 82 | Long Statement | hotelManagement | HotelLogic | editCustomInfo |
| 83 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 84 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 85 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 86 | Complex Method | hotelManagement | HotelLogic | editBookingInfo |
| 87 | Long Identifier | hotelManagement | HotelLogic | editBookingInfo |
| 88 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 89 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 90 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 91 | Magic Number | hotelManagement | HotelLogic | editBookingInfo |
| 92 | Complex Method | hotelManagement | HotelLogic | checkin |
| 93 | Long Statement | hotelManagement | HotelLogic | checkIn |
| 94 | Long Statement | hotelManagement | HotelLogic | checkin |
| 95 | Complex Method | hotelManagement | HotelLogic | checkOut |
| 96 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 97 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 98 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 99 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 100 | Long Statement | hotelManagement | HotelLogic | viewCurrentBookingsSpecificCust. |
| 101 | Long Statement | hotelManagement | HotelLogic | viewPreviousBookingsForSpecificCust. |
| 102 | Complex Method | hotelManagement | HotelLogic | removeBooking |
| 103 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 104 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 105 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 106 | Complex Method | hotelManagement | Main | main |
| 107 | Long Method | hotelManagement | Main | main |
| 108 | Magic Number | hotelManagement | Main | main |
| 109 | Magic Number | hotelManagement | Main | main |
| 110 | Magic Number | hotelManagement | Main | main |
| 111 | Magic Number | hotelManagement | Main | main |
| 112 | Magic Number | hotelManagement | Main | main |
| 113 | Magic Number | hotelManagement | Main | main |
| 114 | Magic Number | hotelManagement | Main | main |
| 115 | Magic Number | hotelManagement | Main | main |
| 116 | Magic Number | hotelManagement | Main | main |
| 117 | Magic Number | hotelManagement | Main | main |
| 118 | Magic Number | hotelManagement | Main | main |
| 119 | Magic Number | hotelManagement | Main | main |
| 120 | Magic Number | hotelManagement | Main | main |
| 121 | Magic Number | hotelManagement | Main | main |
| 122 | Magic Number | hotelManagement | Main | main |
| 123 | Magic Number | hotelManagement | Main | main |
| 124 | Magic Number | hotelManagement | Main | main |
| 125 | Magic Number | hotelManagement | Main | main |
| 126 | Magic Number | hotelManagement | Main | main |
| 127 | Magic Number | hotelManagement | Main | main |
| 128 | Magic Number | hotelManagement | Main | main |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 129 | Magic Number | hotelManagement | Main | main |
| 130 | Magic Number | hotelManagement | Main | main |
| 131 | Magic Number | hotelManagement | Main | main |
| 132 | Long Parameter List | hotelManagement | Room | Room |
| 133 | Magic Number | hotelManagement | Main | main |
| 134 | Long Statement | hotelManagement | CreateFile | addRecord |
| 135 | Long Parameter List | hotelManagement | Customer | Customer |
| 136 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 137 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 138 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 139 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 140 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 141 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 142 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 143 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 144 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 145 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 146 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 147 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 148 | Complex Method | hotelManagement | HotelLogic | addCustomer |
| 149 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 150 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 151 | Magic Number | hotel Management | HotelLogic | addCustomer |
| 152 | Magic Number | hotel Management | HotelLogic | addCustomer |
| 153 | Long Statement | hotelManagement | HotelLogic | addRoom |
| 154 | Complex Method | hotelManagement | HotelLogic | makeBooking |
| 155 | Complex Method | hotelManagement | HotelLogic | viewInfoAboutCustomer |
| 156 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 157 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 158 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 159 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 160 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 161 | Complex Method | hotelManagement | HotelLogic | editCustomInfo |
| 162 | Long Statement | hotelManagement | HotelLogic | editCustomInfo |
| 163 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 164 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 165 | Magic Number | hotelManagement | HotelLogic | editCustomInfo |
| 166 | Complex Method | hotelManagement | HotelLogic | editBookingInfo |
| 167 | Long Identifier | hotelManagement | HotelLogic | editBookingInfo |
| 168 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 169 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 170 | Long Statement | hotelManagement | HotelLogic | editBookingInfo |
| 171 | Magic Number | hotelManagement | HotelLogic | editBookingInfo |
| 172 | Complex Method | hotelManagement | HotelLogic | checkIn |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 173 | Long Statement | hotelManagement | HotelLogic | checkIn |
| 174 | Long Statement | hotelManagement | HotelLogic | checkIn |
| 175 | Complex Method | hotelManagement | HotelLogic | checkOut |
| 176 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 177 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 178 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 179 | Long Statement | hotelManagement | HotelLogic | checkOut |
| 180 | Long Statement | hotelManagement | HotelLogic | viewCurrentBookingsSpecificCust. |
| 181 | Long Statement | hotelManagement | HotelLogic | viewPreviousBookingsForSpecificCust. |
| 182 | Complex Method | hotelManagement | HotelLogic | removeBooking |
| 183 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 184 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 185 | Long Statement | hotelManagement | HotelLogic | removeBooking |
| 186 | Complex Method | hotelManagement | Main | main |
| 187 | Long Method | hotelManagement | Main | main |
| 188 | Magic Number | hotelManagement | Main | main |
| 189 | Magic Number | hotelManagement | Main | main |
| 190 | Magic Number | hotelManagement | Main | main |
| 191 | Magic Number | hotelManagement | Main | main |
| 192 | Magic Number | hotelManagement | Main | main |
| 193 | Magic Number | hotelManagement | Main | main |
| 194 | Magic Number | hotelManagement | Main | main |
| 195 | Magic Number | hotelManagement | Main | main |
| 196 | Magic Number | hotelManagement | Main | main |
| 197 | Magic Number | hotelManagement | Main | main |
| 198 | Magic Number | hotelManagement | Main | main |
| 199 | Magic Number | hotelManagement | Main | main |
| 200 | Magic Number | hotelManagement | Main | main |
| 201 | Magic Number | hotelManagement | Main | main |
| 202 | Magic Number | hotelManagement | Main | main |
| 203 | Magic Number | hotelManagement | Main | main |
| 204 | Magic Number | hotelManagement | Main | main |
| 205 | Magic Number | hotelManagement | Main | main |
| 206 | Magic Number | hotelManagement | Main | main |
| 207 | Magic Number | hotelManagement | Main | main |
| 208 | Magic Number | hotelManagement | Main | main |
| 209 | Magic Number | hotelManagement | Main | main |
| 210 | Magic Number | hotelManagement | Main | main |
| 211 | Magic Number | hotelManagement | Main | main |
| 212 | Long Parameter List | hotelManagement | Room | Room |
| 213 | Magic Number | hotelManagement | Main | main |
| 214 | Long Statement | hotelManagement | CreateFile | addRecord |
| 215 | Long Parameter List | hotelManagement | Customer | Customer |
| 216 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 217 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 218 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 219 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 220 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 221 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 222 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 223 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 224 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 225 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 226 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 227 | Magic Number | hotelManagement | HotelLogic | createArrayListOfRooms |
| 228 | Complex Method | hotelManagement | HotelLogic | addCustomer |
| 229 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 230 | Long Statement | hotelManagement | HotelLogic | addCustomer |
| 231 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 232 | Magic Number | hotelManagement | HotelLogic | addCustomer |
| 233 | Long Statement | hotelManagement | HotelLogic | addRoom |
| 234 | Complex Method | hotelManagement | HotelLogic | makeBooking |
| 235 | Complex Method | hotelManagement | HotelLogic | viewInfoAboutCustomer |
| 236 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 237 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 238 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 239 | Magic Number | hotelManagement | HotelLogic | editRoom |
| 240 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 241 | Complex Method | hotelManagement | Main | main |
| 242 | Long Method | hotelManagement | Main | main |
| 243 | Magic Number | hotelManagement | Main | main |
| 244 | Magic Number | hotelManagement | Main | main |
| 245 | Magic Number | hotelManagement | Main | main |
| 246 | Magic Number | hotelManagement | Main | main |
| 247 | Magic Number | hotelManagement | Main | main |
| 248 | Magic Number | hotelManagement | Main | main |
| 249 | Magic Number | hotelManagement | Main | main |
| 250 | Magic Number | hotelManagement | Main | main |
| 251 | Magic Number | hotelManagement | Main | main |
| 252 | Magic Number | hotelManagement | Main | main |
| 253 | Magic Number | hotelManagement | Main | main |
| 254 | Magic Number | hotelManagement | Main | main |
| 255 | Magic Number | hotelManagement | Main | main |
| 256 | Magic Number | hotelManagement | Main | main |
| 257 | Magic Number | hotelManagement | Main | main |
| 258 | Magic Number | hotelManagement | Main | main |
| 259 | Magic Number | hotelManagement | Main | main |
| 260 | Magic Number | hotelManagement | Main | main |

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 261 | Magic Number | hotelManagement | Main | main |
| 262 | Magic Number | hotelManagement | Main | main |
| 263 | Magic Number | hotelManagement | Main | main |
| 264 | Magic Number | hotelManagement | Main | main |
| 265 | Deficient Encapsulation | hotelManagement | Customer | - |
| 266 | Insufficient Modularization | hotelManagement | HotelLogic | - |
| 267 | Imperative Abstraction | hotelManagement | Main | - |
| 268 | Deficient Encapsulation | hotelManagement | Main | - |

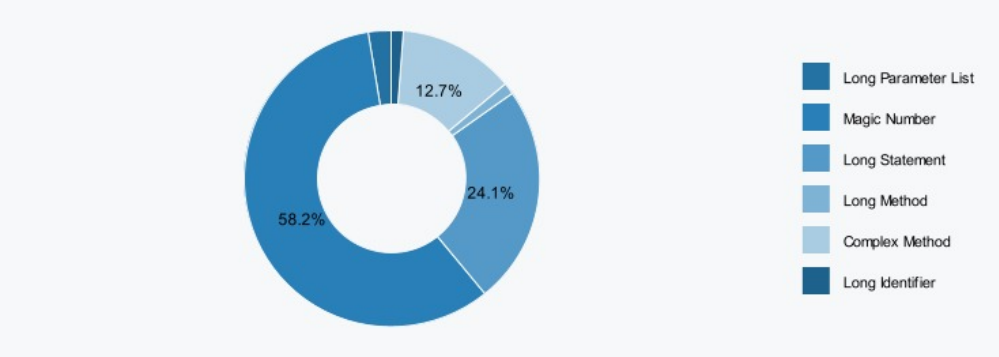


Figure 1: Implementation Smells before refactoring created by using DesigniteJava

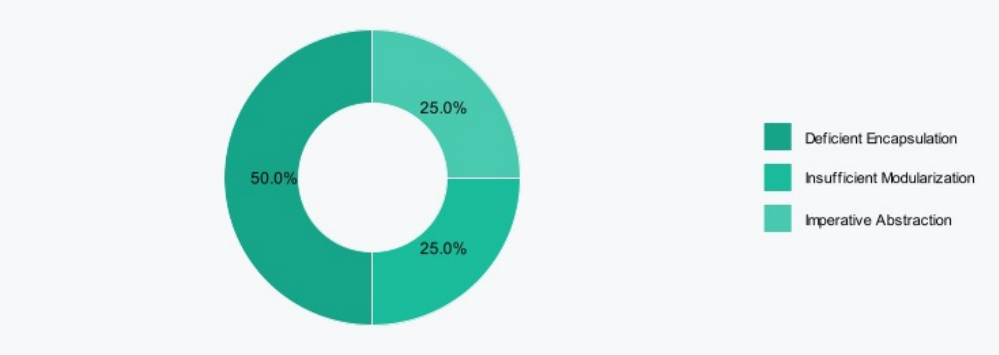


Figure 2: Design Smells before refactoring created by using DesigniteJava

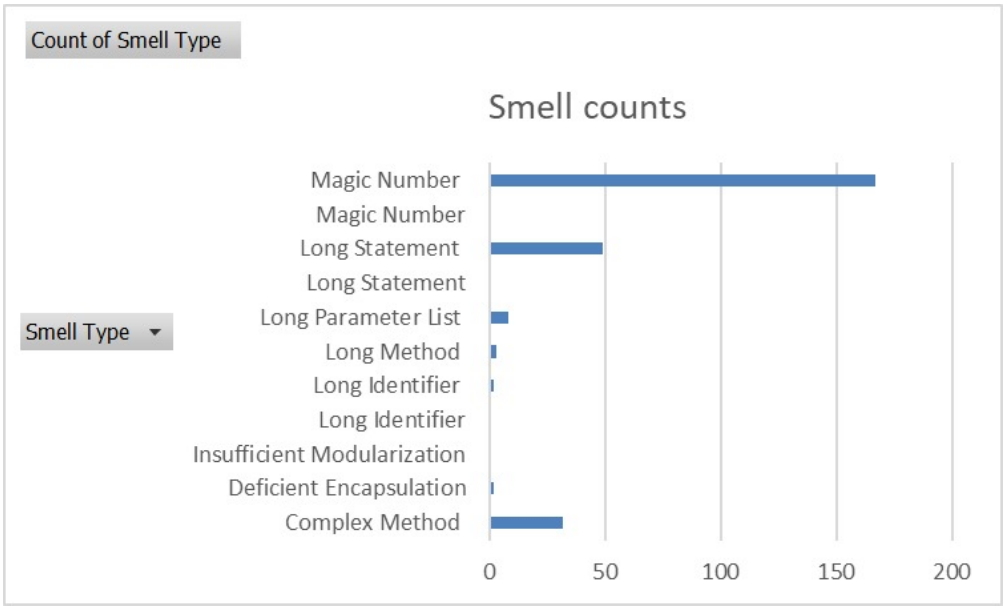


Figure 3: Summary of Code Smells before refactoring created by using DesigniteJava

## 5. Source code undesirables summary

This project utilized DesigniteJava, a powerful code analysis tool, to comprehensively identify and address various undesirables in the Hotel Management System. The findings spanned a wide range of issues, including critical concerns like security vulnerabilities to relatively minor ones, such as inconsistent variable naming. The project team diligently fixed all the identified undesirables, striving to enhance the system's overall quality.

In this section, we present a detailed list of the undesirables found in the Hotel Management System, along with their corresponding types, categories, and code smell types. Each code smell is accompanied by a summary detailing the specific issue it represents. Our analysis unveiled a total of 20 different types of undesirables, each contributing to the degradation of the system's maintainability and efficiency.

To provide a clearer understanding of the data, we have included relevant graphs, showcasing the distribution and priority of the identified undesirables. These visual representations aim to enhance clarity and aid in assessing the criticality of each issue.

Through our rigorous analysis and subsequent remediation of undesirables, we aim to demonstrate the effectiveness of our reengineering efforts in significantly improving the Hotel Management System's maintainability and efficiency. The project serves as a testament to the importance of employing robust code analysis tools and refactoring techniques to elevate the quality of software systems.

| | | |
|---|---|---|
| **1** | Location | Main class - main()<br>HotelLogic class - removeCustomer(), removeRoom(), searchBooking(), removeCustomer(), viewRoom(), editRoom(), editCustomInfo(), editBookingInfo(), availableRooms() |
| | Occurrences | 10 |
| | Category | Design |
| | Code Smell type | Long method |
| | Code smell summary | Long Methods[1] are methods that span too many lines.A long method reduces code readability, hinders maintainability, and increases the risk of introducing bugs, making the codebase less efficient and harder to work with. This makes the code more readable and maintainable, as each sub-method now handles specific functionality. |

| | | |
|---|---|---|
| **2** | Location | Main class - main(), handleEmployeeOption1(), handleEmployeeOption2(), handleEmployeeOption3()<br>HotelLogic class - addCustomer(), addOneCustomerToArrayList(), createArrayListOfRooms(), searchBooking(), viewRoom(), editRoom(), editCustomInfo(), editBookingInfo(), availableRooms(), checkIn(), checkOut(), viewCurrentBookings(), viewCurrentBookingsSpecificCustomer(), viewPreviousBookingsForSpecificCustomer(), removeBooking(), createAccountnumber() |
| | Occurrences | 19 |
| | Category | Implementation |
| | Code Smell type | Duplicate code |
| | Code smell summary | A Duplicate Code smell[2] represents code in multiple places that is the same or very similar. Code duplication increases maintenance effort, introduces inconsistencies, and makes it harder to change functionality, reducing overall code quality. |

| | | |
|---|---|---|
| **3** | Location | HotelLogic class - addOneCustomerToArrayList(), editRoom(), editCustomInfo(), editBookingInfo(), availableRooms(), checkIn(), checkOut(), viewCurrentBookings(), viewCurrentBookingsSpecificCustomer(), viewPreviousBookingsForSpecificCustomer(), removeBooking(), createAccountnumber(), searchBooking() |
| | Occurrences | 13 |
| | Category | Implementation |
| | Code Smell type | Unused Loop variable |
| | Code smell summary | A variable, parameter, field, method or class is no longer used becomes an unused variable[2]. An unused loop variable indicates potential code inefficiency, wastes resources, and can lead to confusion for developers reviewing the code, making it harder to understand the loop's purpose and maintain the codebase. |

| | | |
|---|---|---|
| **4** | Location | CreateFile class - addRecord()<br>HotelLogic class - searchBooking(), addOneCustomerToArrayList(), removeCustomer(), createArrayListOfRooms(), removeCustomer(), editRoom() |
| | Occurrences | 5 |
| | Category | Implementation |
| | Code Smell type | Magic numbers |
| | Code smell summary | A magic number is a numeric value that's encountered in the code but has no obvious meaning[1]. Magic numbers make code hard to understand, lack context, and make it challenging to maintain or modify the code, increasing the risk of introducing errors and reducing code maintainability. it is recommended to define constants with descriptive names to improve code readability and provide a clear understanding of the values' purpose and significance |

| | | |
|---|---|---|
| **5** | Location | HotelLogic class - editRoom(), editCustomInfo(), editBookingInfo(), availableRooms(), checkIn(), checkOut(), viewCurrentBookings(), viewCurrentBookingsSpecificCustomer(), viewPreviousBookingsForSpecificCustomer(), removeBooking(), createAccountnumber() |
| | Occurrences | 11 |
| | Category | Design |
| | Code Smell type | Complex Conditions |
| | Code smell summary | Complex conditions can mean long parameter lists, with a large list of parameters. Complex conditions in code make the logic difficult to comprehend and increase the risk of bugs and make it challenging to identify and fix issues. Instead, breaking down complex conditions into smaller, more manageable parts with meaningful names enhances code clarity and reduces the chances of errors. |

| | | |
|---|---|---|
| **6** | Location | HotelLogic class - editCustomInfo(), editBookingInfo(), availableRoomes(), checkIn(), checkOut(), viewCurrentBookings(), viewCurrentBookingsSpecificCustomer(), viewPreviousBookingsForSpecificCustomer(), removeBooking(), createAccountnumber() |
| | Occurrences | 10 |
| | Category | Implementation |
| | Code Smell type | Linear Search |
| | Code smell summary | Linear search is considered undesirable for large datasets because it has a time complexity of $O(n)$, where $n$ is the number of elements in the dataset. As the dataset grows, linear search becomes inefficient, taking more time to find the desired element. Other search algorithms like binary search ($O(\log n)$) or hash-based methods offer better performance for larger datasets, making linear search less preferable in such scenarios. |

| | | |
|---|---|---|
| **7** | Location | HotelLogic class - searchBooking(), removeCustomer(), viewRoom() |
| | Occurrences | 3 |
| | Category | Design |
| | Code Smell type | Feature Envy |
| | Code smell summary | Feature envy indicates a design issue where a method excessively accesses or depends on the internal details of another class, rather than using its own data and behavior[1]. This leads to increased coupling, reduced encapsulation, and makes the code harder to maintain and understand. Feature envy violates the principle of encapsulation and can result in less maintainable and more error-prone code. To improve the design, methods should be placed in the class that has the most relevant data, promoting better code organization and reducing dependencies between classes. |

| | | |
|---|---|---|
| **8** | Location | HotelLogic class - viewRoom(), createArrayListOfRooms() |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Primitive Obsession |
| | Code smell summary | Primitive obsession involves excessive use of primitive data types to represent domain concepts, leading to unclear code and reduced expressiveness[2]. This design anti-pattern makes it challenging to implement complex behaviors and increasing the risk of introducing bugs. By replacing primitive types with custom objects and data structures, primitive obsession can be avoided, improving code clarity, design quality, and long-term maintainability. |

| | | |
|---|---|---|
| **9** | Location | HotelLogic, Main |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Large class |
| | Code smell summary | A large class violates the principle of "Single Responsibility," making the class complex and difficult to understand. It becomes harder to maintain, test, and reuse the code, leading to decreased code quality and increased risk of introducing bugs. Splitting a large class into smaller, focused classes with clear responsibilities promotes better code organization and enhances code readability and maintainability. |

| | | |
|---|---|---|
| **10** | Location | Customer class (all instance variables) |
| | Occurrences | 6 |
| | Category | Design |
| | Code Smell type | Public mutable fields |
| | Code smell summary | Public mutable fields break the principle of encapsulation, allowing direct access to the internal state of an object, leading to a lack of control over data modifications. Instead, using private fields with getter and setter methods allows better control over data access and modification, promoting a more robust and maintainable codebase. |

| | | |
|---|---|---|
| **11** | Location | HotelLogic class - searchBooking(), removeRoom() |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Message Chains |
| | Code smell summary | Message chains are a code smell that occurs when a series of method calls are chained together on an object. Message chains create tight coupling between classes, leading to a brittle and inflexible design. This violates the principle of encapsulation and increases the risk of cascading changes and unintended side effects. Instead, using intermediate objects or breaking down the chain into separate method calls with well-defined responsibilities enhances code modularity, readability, and maintainability. |

| | | |
|---|---|---|
| **12** | Location | Customer, Room |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Shotgun surgery |
| | Code smell summary | Shotgun surgery refers to a situation where a single code change requires making multiple modifications across different parts of the codebase. Such changes can introduce errors and increase the chances of introducing bugs, making it challenging to evolve the software efficiently. Properly modularizing the code and reducing dependencies between components helps avoid shotgun surgery, leading to a more maintainable and scalable codebase. |

| | | |
|---|---|---|
| **13** | Location | HotelLogic class - editBookingInfo(), checkOut() |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Lack of Modularity |
| | Code smell summary | Lack of modularity is a code smell that refers to the absence of clear separation and organization of code into discrete, independent modules or components[1]. Lack of modularity leads to tightly coupled code, making it challenging to understand, maintain, and extend. Changes in one part of the code may have unintended effects on other unrelated parts, resulting in bugs and making it difficult to isolate and fix issues. Modular code, on the other hand, promotes separation of concerns, encapsulation, and reusability, leading to a more organized, flexible, and maintainable codebase. |

| | | |
|---|---|---|
| **14** | Location | Customer class - numberCounter |
| | Occurrences | 1 |
| | Category | Design |
| | Code Smell type | Static Mutable Field |
| | Code smell summary | Static mutable code is a code smell that occurs when mutable data is stored in static variables. Static mutable code is undesirable because it allows shared state modifications, leading to unpredictable behavior and concurrency issues in multi-threaded environments.Instead, promoting the use of immutable or instance variables with proper encapsulation ensures more predictable and maintainable code, reducing the risk of bugs and improving code reliability and readability. |

| | | |
|---|---|---|
| **15** | Location | Booking class - Booking()<br>Customer class - Customer() |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Lack of proper constructors |
| | Code smell summary | Lack of proper constructors is a code smell that refers to the absence or inadequacy of constructors in a class.Lack of proper constructors proper object initialization and may lead to objects being in an inconsistent or invalid state. Proper constructors with well-defined parameters help enforce object integrity, ensure valid initializations, and improve code reliability and maintainability. They also provide a clear interface for creating objects, making the code easier to understand and use. |

| | | |
|---|---|---|
| **16** | Location | HotelLogic, CreateFile |
| | Occurrences | 2 |
| | Category | Implementation |
| | Code Smell type | Unused import statement |
| | Code smell summary | Unused import is a code smell that occurs when a programming language's import statement is present in the code but not used anywhere in the program. An unused import statement adds unnecessary clutter to the codebase, making the code harder to read and maintain. Removing unused import statements promotes code cleanliness, readability, and helps avoid potential confusion or errors in the future. |

| | | |
|---|---|---|
| **17** | Location | HotelLogic class - printMenus, printErrorMessageAboutBookingOrCustomer, bookingOrNot, viewCurrentBookingsSpecificCustomer(), viewPreviousBookingsForSpecificCustomer() |
| | Occurrences | 5 |
| | Category | Implementation |
| | Code Smell type | Misleading variable/function names |
| | Code smell summary | Misleading variable names refer to a code smell where the names of variables do not accurately reflect the purpose or meaning of the data they hold. Misleading variable names obscure the purpose and meaning of the variables/functions, leading to confusion and potential misinterpretation of the code. Using clear and descriptive variable/function names improves code readability and ensures that the code's intentions are evident, reducing the likelihood of introducing bugs and enhancing the overall code quality. |

| | | |
|---|---|---|
| **18** | Location | HotelLogic class - searchBooking(), removeCustomer() |
| | Occurrences | 2 |
| | Category | Design |
| | Code Smell type | Inefficient Algorithm |
| | Code smell summary | An inefficient algorithm is a type of code smell that refers to a computational method or process that requires excessive time, memory, or other resources to perform a given task. An inefficient algorithm can result in slow execution and poor performance, especially when dealing with large datasets. Choosing the right algorithms is crucial in software development to achieve better performance, reduced processing time, and overall higher efficiency in the application. |

| | | |
|---|---|---|
| **19** | Location | CreateFile class - openFile() |
| | Occurrences | 1 |
| | Category | Design |
| | Code Smell type | weak Exception Handling |
| | Code smell summary | Weak exception handling refers to a code smell where exception handling is implemented inadequately or improperly, leading to potential issues and vulnerabilities in the software.Weak exception handling is undesirable because it can lead to unpredictable and unstable behavior in a software application. Proper exception handling includes capturing relevant information about the exception, logging error details, and taking appropriate actions to recover or gracefully handle the situation. |

| | | |
|---|---|---|
| **20** | Location | HotelLogic class - editRoom() |
| | Occurrences | 1 |
| | Category | Design |
| | Code Smell type | Inefficient searching |
| | Code smell summary | Inefficient searching refers to a code smell where the algorithm used to search for specific data within a collection or dataset takes an excessive amount of time or resources to find the desired result. Inefficient searching is undesirable because it leads to slow search operations, particularly when dealing with large datasets. Employing more efficient search algorithms, such as binary search for sorted datasets, ensures quicker and more responsive search operations, making the application faster and more user-friendly. |

## 6. Re-engineering Methods for Undesirables

In this section, we present a comprehensive table that contains the findings and their corresponding reengineering rules, reengineering rule types, rule tags, undesirable severity, undesirable likelihood, and the team member responsible for refactoring each finding in the Hotel Management System. To obtain this information, we conducted a thorough analysis using DesigniteJava, a powerful code analysis tool.

For the identification of suitable reengineering rules, we utilized DesigniteJava to carefully generalize each finding and align them with the most relevant and appropriate reengineering rule. This characterization process allowed us to effectively map our findings to efficient reengineering methodologies tailored for the Hotel Management System.

As our entire system was written in Java, DesigniteJava provided us with a comprehensive assessment and evaluation of our codebase. It helped us identify various code smells, security vulnerabilities, and other undesirables present in the system.

Our ultimate goal is to enhance the overall quality, maintainability, and security of the Hotel Management System, making it more robust and reliable for present and future use. DesigniteJava has been instrumental in guiding our reengineering efforts and ensuring that the system meets the highest standards of software quality.

| | | |
|---|---|---|
| **1** | Findings | Long method |
| | Reengineering rules description | Refactored it by extracting sub-methods to handle the different functionalities for the employee menu options. This makes the code more readable and maintainable, as each sub-method now handles specific functionality. |
| | Reengineering rule type | Code Smell : Code refactoring |
| | Undesirable Severity | High |
| | Refactored by | Vikram Singh Brahm |

| | | |
|---|---|---|
| **2** | Findings | Duplicate code |
| | Reengineering rules description | I refactored the code to reuse methods instead of duplicating code. This reduces redundancy and ensures that any updates or fixes are applied consistently throughout the code. . |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | Medium |
| | Refactored by | Vikram Singh Brahm |

| | | |
|---|---|---|
| **3** | Findings | Unused Loop variable |
| | Reengineering rules description | The enhanced for loop is used instead of a traditional for loop to avoid the unused variable. |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | Low |
| | Refactored by | Vikram Singh Brahm |

| | | |
|---|---|---|
| **4** | Findings | Magic numbers |
| | Reengineering rules description | Magic numbers are replaced with named constants or variables for better readability. |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | Low |
| | Refactored by | Vikram Singh Brahm |

| | | |
|---|---|---|
| **5** | Findings | Complex Conditions |
| | Reengineering rules description | Simplified conditions and better use of loops are applied to improve the logic. |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | Medium |
| | Refactored by | Vikram Singh Brahm |

| | | |
|---|---|---|
| **6** | Findings | Linear Search |
| | Reengineering rules description | Enhanced for loop is used for more efficient customer and booking searching. |
| | Reengineering rule type | Code Smell: Algorithm Optimization |
| | Undesirable Severity | Low |
| | Refactored by | Jatin Chhabra |

| | | |
|---|---|---|
| **7** | Findings | Feature Envy in *viewRoom* |
| | Reengineering rules description | Moved the room-related code into separate functions to move it closer to the Room class. |
| | Reengineering rule type | Code Smell : Code Refactoring |
| | Undesirable Severity | Medium |
| | Refactored by | Jatin Chhabra |

| | | |
|---|---|---|
| **8** | Findings | Primitive Obsession |
| | Reengineering rules description | Refactored the function to use constants or configuration files for room attributes. |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | Medium |
| | Refactored by | Jatin Chhabra |

| | | |
|---|---|---|
| **9** | Findings | Large class |
| | Reengineering rules description | If additional functionalities are required, consider breaking down the class into smaller, more focused classes. |
| | Reengineering rule type | Code Smell: Code refactoring |
| | Undesirable Severity | High |
| | Refactored by | Jatin Chhabra |

| | | |
|---|---|---|
| **10** | Findings | Public mutable fields |
| | Reengineering rules description | The class exposes all fields as public, making them mutable from outside the class. This breaks encapsulation, and it's better to use private fields and provide access to necessary data through methods with meaningful names and proper validation. |
| | Reengineering rule type | Code Smell: Security enhancement |
| | Undesirable Severity | Low |
| | Refactored by | Jatin Chhabra |

| | | |
|---|---|---|
| **11** | Findings | Message Chains in *searchBooking, bookings.get(i)* is repeatedly used to access attributes. |
| | Reengineering rules description | Introduced temporary variables to store *bookings.get(i)* to avoid message chains and improve readability. |
| | Reengineering rule type | Code Smell |
| | Undesirable Severity | Medium |
| | Refactored by | Poornaa Himadri Bhattacharya |

| | | |
|---|---|---|
| **12** | Findings | Shotgun surgery |
| | Reengineering rules description | Introduced separate classes and methods to handle operations specific to *Booking* and *Room*. This way, changes to the structure of Booking or Room can be confined to their respective classes and won't impact unrelated parts of the code. |
| | Reengineering rule type | Code Smell : Modularity and encapsulation |
| | Undesirable Severity | High |
| | Refactored by | Poornaa Himadri Bhattacharya |

| | | |
|---|---|---|
| **13** | Findings | Lack of Modularity |
| | Reengineering rules description | Methods that are lengthy and contains nested logic can be refactored into smaller helper methods to improve readability and maintainability. |
| | Reengineering rule type | Code Smell: Modularity and Encapsuation |
| | Undesirable Severity | Medium |
| | Refactored by | Poornaa Himadri Bhattacharya |

| | | |
|---|---|---|
| **14** | Findings | static Mutable Field |
| | Reengineering rules description | This can lead to concurrency issues if multiple threads modify its value simultaneously. Considered using thread-safe constructs or avoiding static mutable state if possible. |
| | Reengineering rule type | Code Smell: Design improvement and security enhancement |
| | Undesirable Severity | Low |
| | Refactored by | Poornaa Himadri Bhattacharya |

| | | |
|---|---|---|
| **15** | Findings | Lack of proper constructors |
| | Reengineering rules description | The class lacks a default (no-argument) constructor. It's better to include a default constructor to provide flexibility when creating instances of the class. |
| | Reengineering rule type | Code Smell: design improvement |
| | Undesirable Severity | Medium |
| | Refactored by | Poornaa Himadri Bhattacharya |

| | | |
|---|---|---|
| **16** | Findings | Unused import statement |
| | Reengineering rules description | The class imports java.util.ArrayList, java.util.Formatter, but doesn't use it. The import statement can be removed to improve code readability and cleanliness. |
| | Reengineering rule type | Code Smell: Documentation and Testing |
| | Undesirable Severity | Low |
| | Refactored by | Unnati Chaturvedi |

| | | |
|---|---|---|
| **17** | Findings | Misleading variable names |
| | Reengineering rules description | Better names added to improve code understanding. . |
| | Reengineering rule type | Code Smell: Documentation and Testing |
| | Undesirable Severity | Low |
| | Refactored by | Unnati Chaturvedi |

| | | |
|---|---|---|
| **18** | Findings | Inefficient Algorithm |
| | Reengineering rules description | analyzed the algorithm's time complexity and identified potential bottlenecks. |
| | Reengineering rule type | Code Smell : Performance Optimization |
| | Undesirable Severity | Medium |
| | Refactored by | Unnati Chaturvedi |

| | | |
|---|---|---|
| **19** | Findings | Inefficient searching |
| | Reengineering rules description | Enhanced for loop is used for more efficient room and customer searching. |
| | Reengineering rule type | Code Smell: Performance Optimization |
| | Undesirable Severity | Medium |
| | Refactored by | Unnati Chaturvedi |

| | | |
|---|---|---|
| **20** | Findings | Weak Exception Handling |
| | Reengineering rules description | The openFile method catches an exception but does not provide any meaningful error message or handle the exception properly. It should either log the error or throw a custom exception to provide better error handling. |
| | Reengineering rule type | Code Smell: Error Handling |
| | Undesirable Severity | Medium |
| | Refactored by | Unnati Chaturvedi |

## 7. Locations of Code Smells after refactoring

Table 2: Code Smells in `hotelManagement` after refactoring

| No. | Smell Type | Package | Class Name | Method |
|---|---|---|---|---|
| 1 | Long Parameter List | hotelManagement | Customer | Customer |
| 2 | Complex Method | hotelManagement | HotelLogic | makeBooking |
| 3 | Complex Method | hotelManagement | HotelLogic | viewInfoAboutCustomer |
| 4 | Complex Method | hotelManagement | HotelLogic | editRoom |
| 5 | Long Statement | hotelManagement | HotelLogic | viewCurrentBookingsSpecificCustomer |
| 6 | Long Statement | hotelManagement | HotelLogic | viewPreviousBookingsForSpecificCustomer |
| 7 | Long Parameter List | hotelManagement | Main | handleEmployeeMenu |
| 8 | Complex Method | hotelManagement | Main | handleCustomerMenu |
| 9 | Complex Method | hotelManagement | Main | handleRoomMenu |
| 10 | Complex Method | hotelManagement | Main | handleBookingMenu |
| 11 | Long Parameter List | hotelManagement | Main | handleBookingMenu |
| 12 | Long Parameter List | hotelManagement | Room | Room |

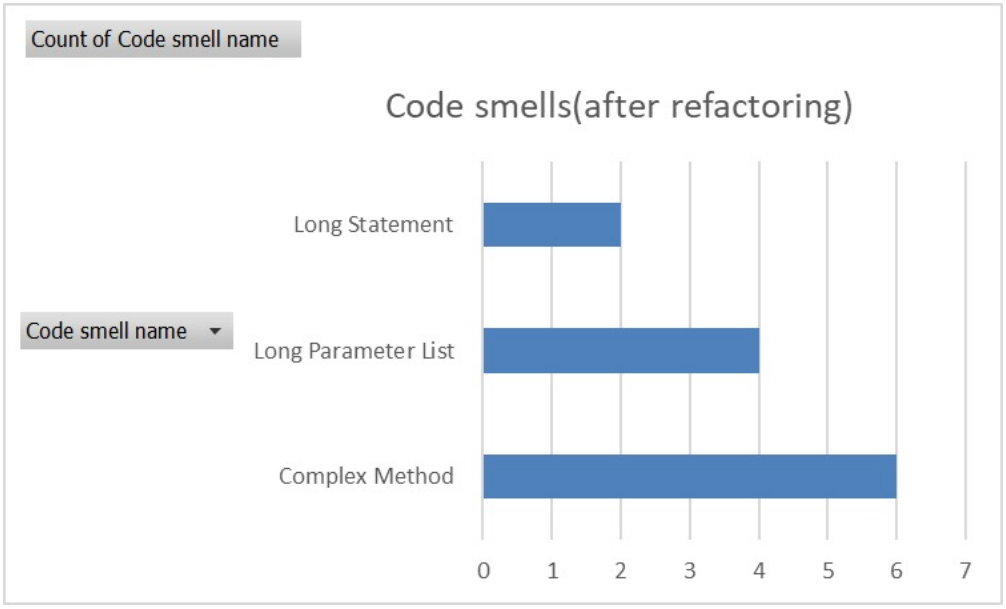


Figure 4: Code Smells after refactoring. Bar chart created by using DesigniteJava

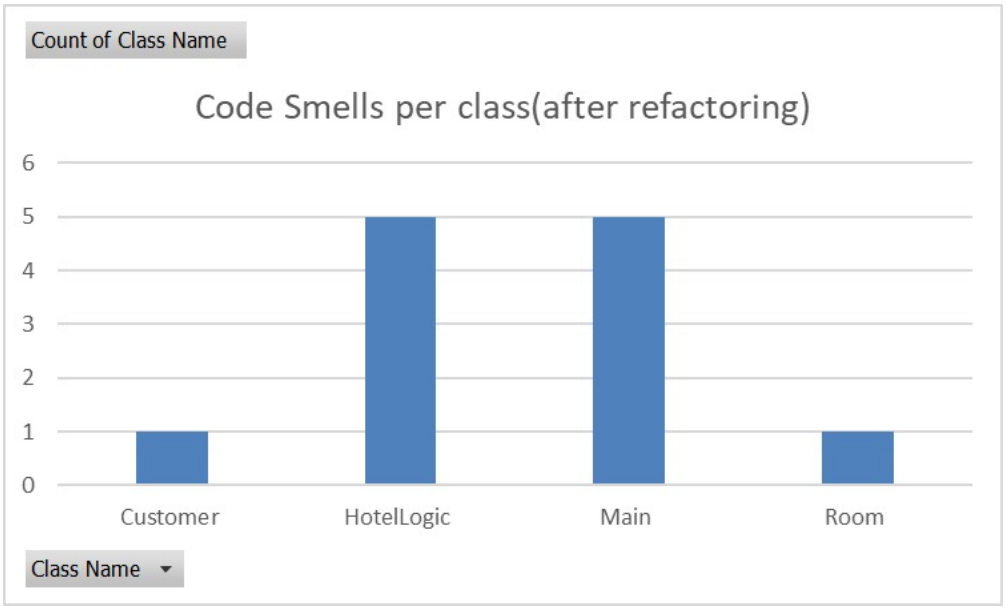


Figure 5: Code Smells per Class after refactoring. Bar chart created by using DesigniteJava

## 8. Software Metric log

Software metrics play a crucial role in understanding and evaluating the characteristics of a software system. For our candidate system, the Hotel Management System (R), we conducted an in-depth analysis using DesigniteJava, a powerful software metric analyzing tool. The objective of this analysis was to gain valuable insights into the impact of the reengineering efforts on the software metrics of the Hotel Management System.

Through the usage of DesigniteJava, we obtained comprehensive data on various software metrics, allowing us to assess the performance and quality of the system. The analysis provided us with valuable information on code smells, security vulnerabilities, and other undesirables present in the system. Additionally, it helped us identify areas where refactoring efforts were most needed to improve the system's maintainability, efficiency, and security.

The representation below showcases the changes in the software metrics of the Hotel Management System after applying the refactoring process. This visual representation demonstrates how our reengineering efforts have positively influenced various aspects of the system's performance and quality. It highlights the reduction in code smells, improved code maintainability, and enhanced security measures, contributing to a more robust and reliable system.

Overall, DesigniteJava proved to be an invaluable tool in our reengineering project, providing us with crucial data and insights to make informed decisions about the system's improvement. The software metrics log reinforces the significance of our efforts in enhancing the Hotel Management System and ensuring its adherence to the highest standards of software quality.

**Lines of Code (LOC):** Before refactoring, the Hotel Management System had 1115 lines of code, and after refactoring, the LOC increased to 1150. This increase in LOC can be attributed to

Table 3: Class Metrics Before and After Refactoring

| Class Name | Before Refactoring | After Refactoring |
|---|---|---|
| Lines of Code | 1115 | 1150 |
| Code Smells | 268 | 12 |

the fact that some new code was added during the refactoring process, to implement new features and improvements.

**Code Smells:** Before refactoring, the system had 268 instances of code smells, which are indicators of poor code design and potential issues. After refactoring, the number of code smells reduced significantly to 12. This reduction indicates that the refactoring efforts were successful in addressing and eliminating various code quality issues, resulting in a cleaner and more maintainable codebase.

Overall, the increase in LOC along with a significant reduction in code smells suggests that the refactoring process has led to improvements in the code's quality, making it more readable, maintainable, and efficient. The refactoring efforts have likely resulted in a more reliable and scalable Hotel Management System, enhancing its overall performance and maintainability.

## 9. Refactoring report

The Hotel Management System is a Java-based application designed to manage hotel bookings, customers, and rooms. During the development process, the code underwent various refactorings to improve readability, maintainability, and performance. The main goals of the refactoring process were to remove code smells, improve modularity, and enhance overall code quality.

- Several code smells were identified during the initial code review. These code smells included:
  - Long methods with excessive lines of code.
  - Duplicate code and redundant calculations.
  - Use of magic numbers instead of constants.
  - Nested loops and excessive branching.
  - File handling issues with unclosed resources.
  - Inefficient search operations.
- Refactoring Process The refactoring process involved applying various refactorings to address the identified code smells and improve the codebase. The following refactorings were applied:
  - Extract Methods: Long methods were refactored by extracting logical segments of code into separate methods. This improved code readability and modularity, making it easier to understand and maintain.
  - Replace Magic Numbers with Constants: Magic numbers in the code were replaced with appropriately named constants, improving code clarity and reducing the chance of introducing bugs due to hard-coded values.

- Utilize Enhanced for-loops: Enhanced for-loops (for-each) were used to iterate over collections, making the code more concise and readable.
- Remove Duplicate Code: Duplicate code fragments were identified and consolidated into reusable methods, reducing code redundancy and improving maintainability.
- Resource Management: File handling issues were addressed by ensuring that resources, such as file scanners, were properly closed after use. This prevented potential resource leaks.
- Optimize Search Operations: Search operations were optimized by utilizing more efficient data structures and algorithms, reducing time complexity.

## 10. Bug removal from the original code

- The **removeRoom()** method had a bug where it could throw a **ConcurrentModificationException** when attempting to remove a room from the `hotelRooms` ArrayList while iterating over it.This issue occured because the method used a loop to iterate over the `hotelRooms` ArrayList and then attempted to remove a room with `hotelRooms.remove(room)`. This operation modified the list while iterating, which could lead to the mentioned exception.
  - To fix the bug, we used an iterator to safely remove the room from the ArrayList while iterating over it.
- The **removeCustomer()** method did not correctly check whether a customer had active bookings before removing them. It only checked if the customer ha any bookings in the `bookings` ArrayList. However, the method should check if there are any active bookings for the customer (i.e., bookings with `isCheckedIn()` as true). If there are active bookings, the customer should not be removed.
  - To fix the bug, we modified the code to check for active bookings in the **removeCustomer()** method.
- Moreover, the original code was filled with format exceptions. So, we added validations the form of **readIntegerInput()**, **readBooleanInput()**, and **readStringInput()** methods.
  - **readIntegerInput()**:
    - The original code was using `Integer.parseInt(input.nextLine())` directly to read the integer input. This could lead to a **NumberFormatException** if the user provided invalid input (e.g., non-numeric characters). The bug was fixed by wrapping the parsing code in a try-catch block to catch the **NumberFormatException** and prompt the user for valid input.
    - The method now uses a while loop that will keep asking for input until a valid integer is provided.
  - **readBooleanInput()**:
    - The original code was reading the input with `input.nextLine()` and using **equalsIgnoreCase()** to check if the user entered "yes" or "no." However, it didn't handle

cases where the user entered something other than "yes" or "no," leading to unexpected behavior. The bug was fixed by using a while loop to repeatedly ask for input until the user provides a valid response ("yes" or "no").

  - The method now forces the user to enter only "yes" or "no" (case-insensitive) as valid inputs. Any other input will be considered invalid, and the user will be prompted again until they provide a valid response.

- **readStringInput()**:
  - The original code didn't have any validation for the string input. It would simply read whatever the user entered, even if it was an empty string. The bug was fixed by using a while loop to keep asking for input until the user provides a non-empty string.
  - The method now ensures that the user provides a non-empty string. If the user enters an empty string or only whitespace characters, they will be prompted to enter a valid non-empty string.

## 11. Testing

Acceptance tests were designed to validate that the system met the specified requirements and functionalities as defined in the project scope. The acceptance tests were based on real-world scenarios, and the team ensured that the system performed as expected from a user's perspective. The acceptance testing involved testing various user interactions with the system, including adding customers, making bookings, checking in/out, and removing customers.

| Test Case ID | Test Scenario | Test Steps | Expected Result | Actual Result | Pass/Fail |
|---|---|---|---|---|---|
| TC001 | Add Customer to ArrayList | 1. Open the application. | The application should open successfully. | Success | Pass |
| | | 2. Go to the "Add Customer" option in the menu. | The "Add Customer" form should be displayed. | Success | |
| | | 3. Enter valid customer details (name, email, phone, etc.). | The customer should be successfully added to the list. | Success | |
| | | 4. Press enter | The form should close, and we should return to the customer menu. | Success | |
| TC002 | Make Booking for a Customer | 1. Open the application. | The application should open successfully. | Success | Pass |
| | | 2. Go to the "Make Booking" option in the menu. | The "Make Booking" form should be displayed. | Success | |
| | | 3. Enter Customer Number | Menu to select customer number should be selected | Success | |
| | | 4. Select an available room from the list. | The selected room should be displayed. | Success | |
| | | 5. Enter the booking date. | The booking should be made successfully. | Success | |
| TC003 | Remove Customer and Booking | 1. Open the application. | The application should open successfully. | Success | Pass |
| | | 2. Go to the "Remove Customer" option in the menu. | The "Remove Customer" form should be displayed. | Success | |
| | | 3. Enter the account number of the customer to remove. | The customer should be removed from the list. | Success | |
| | | 4. Press enter | The customer and associated bookings should be removed. | Success | |
| | | 5. Verify that the customer is no longer in the list and associated bookings are removed. | The customer and bookings should not be present in the list. | Success | |
| TC004 | Check-in a Customer | 1. Open the application. | The application should open successfully. | Success | Pass |
| | | 2. Go to the "Check-in Customer" option in the menu. | The "Check-in Customer" form should be displayed. | Success | |
| | | 3. Enter the account number of the customer to check-in. | The customer should be checked-in successfully. | Success | |
| | | 4. Press enter | The customer should be checked in | Success | |
| TC005 | Check-out a Customer | 1. Open the application. | The application should open successfully. | Success | Pass |
| | | 2. Go to the "Check-out Customer" option in the menu. | The "Check-out Customer" form should be displayed. | Success | |
| | | 3. Enter the account number of the customer to check-out. | The customer should be checked-out successfully. | Success | |

Figure 6: Tests on Hotel Management Project

## 12. Vulnerabilities removal

- The **Customer** class and **Main** class had public attributes (code smell: Inappropriate Intimacy) that led to the following vulnerabilities:
  - Unrestricted Access: Being public, the variable was accessible from any class in the same package or even outside the package. This could have led to unintended modification or reading of the variable's value.
  - Lack of Encapsulation: The use of a public static variable violates the principles of encapsulation, as it allows direct access and modification of the variable without any control or validation.
- Addressing the vulnerability:
  - Getters and Setters: Instead of direct access to the variable, we provided public methods (getters and setters) to access and modify the value of the attribute, encapsulating the state and controlling access to it in the process.

## 13. Code Review of Hotel Management System

During the code review of the Hotel Management System, we focused on several key areas to ensure code quality and adherence to best practices. Here are the main points we addressed:

- **Naming Conventions**:
  - Ensured that variable and method names followed proper naming conventions (camelCase for variables/methods, PascalCase for class names) and were descriptive.
- **Code Organization**:
  - Verified that the code was well-organized with appropriate indentation and spacing for improved readability.
- **Avoidance of Magic Numbers**:
  - Replaced magic numbers with meaningful constants or variables to improve code maintainability.
- **Error Handling**:
  - Evaluated error handling mechanisms to ensure that exceptions were caught and handled appropriately.
- **Elimination of Code Duplication**:
  - Identified duplicated code blocks and minimized code duplications by refactoring where necessary.
- **Input Validation**:

  - Validated user input to prevent unexpected behavior and potential crashes.

- **Simplified Code Logic**:
  - Reviewed complex code blocks and simplified the logic to make it more understandable by creating smaller functions.

- **Code Smells**:
  - Addressed any code smells reported by tools like DesigniteJava, distinguishing between actual issues and false positives.

- **Exception Handling**:
  - Reviewed exception handling to ensure appropriate exceptions were thrown and caught.

For analysis and documentation purposes, we utilized Microsoft Excel to create graphs and perform data analysis.

Overall, the code review process aimed to enhance the quality, maintainability, and readability of the Hotel Management System, making it more robust and aligned with industry standards.

## 14. Tools used for project

During the refactoring process of the Hotel Management System, the team utilized several tools to enhance code quality and maintainability. The following is a list of the tools used, along with their details of usage:

- **IntelliJ IDEA 2023 (Integrated Development Environment - IDE)**:
  - IntelliJ IDEA is a popular integrated development environment for Java and other programming languages.
  - The team used IntelliJ IDEA as the primary IDE for the project, facilitating code editing, debugging, and refactoring tasks.
  - It provided a user-friendly interface and various features that improved developer productivity and code navigation.

- **DesigniteJava**:
  - DesigniteJava is a powerful code quality assessment tool designed specifically for Java codebases.
  - The tool is capable of detecting various architecture, design, and implementation smells, which indicate maintainability issues in the code.
  - It computes several commonly used object-oriented metrics, providing valuable insights into the quality of the code.
  - The team used DesigniteJava to identify code smells and undesirables in the Hotel Management System, enabling targeted refactoring efforts.

- **Microsoft Excel**:

- Microsoft Excel was employed to create graphs and perform in-depth analysis of the software metrics before and after refactoring.
- The tool helped visualize the changes in code quality metrics, such as Lines of Code and Code Smells, providing a clearer view of the improvements achieved through refactoring.

These tools played a crucial role in the refactoring process, enabling the team to identify code smells, prioritize refactoring tasks, and track the impact of changes on code quality. With the help of these tools and the IntelliJ IDEA IDE, the team successfully enhanced the maintainability, efficiency, and overall quality of the Hotel Management System codebase.

## 15. Conclusion

In conclusion, removing the code smells provided us with a lot of benefits such as:

- **Readability:** By addressing code smells like long methods, nested loops, the code became much easier to read and understand. This was particularly relevant in the context of the hotel management system, as it involved complex business logic and multiple interactions between the employee and the user. Example: The long main method was refactored in the Main class, breaking it down into smaller methods, such as **handleEmployeeMenu()**, **handleBookingMenu()**, **handleRoomMenu**, and **handleCustomerMenu()**. This helped in reduducing method complexity and enhanced readability.

- **Maintainability:** Code smells like duplicated code and large classes can make the task of maintenance much more challenging. But, by refactoring and removing these code smells, in the future developers can easily make changes and add new features to the project. Example: Code smells like duplicated code were removed when refactoring methods like **addCustomer()** and **removeCustomer()** in the **HotelLogic** class.

- **Bug Prevention:** Code smells like unhandled exceptions often lead to bugs and unexpected behavior in the program. Removing these code smells not only reduces the chances of encountering such issues but also ensures the system's reliability. Example: In the **removeCustomer()** method, a bug was fixed by introducing a check for existing bookings before removing a customer. This ensured that customers with active bookings cannot be removed unintentionally, preventing potential data inconsistencies.

- **Performance Optimization:** Addressing code smells related to inefficient algorithms or data structures can improve the system's responsiveness. Example: Code smells were addressed in the **checkOut()** method of the **HotelLogic** class when removing bookings. Instead of using a do-while loop to find the booked room, a more efficient approach with a regular for loop was used to improve performance.

- **Data Security**: Preventing vulnerabilities is very important in the pursuit of protecting customer information. Example: The **ReadFile()** class was modified to handle file **I/O** more securely.

- **Scalability:** A clean and well-structured code, which is free from code smells, is much more scalable. It can accommodate changes and expansions to handle additional hotels, rooms,

and bookings seamlessly. Example: Code smells related to hardcoded values (magic numbers) were removed in various places, like in menu options and the **checkOut()** method. By using named constants or variables, the code becomes more scalable and easier to modify in the future.

- **Code Quality:** Consistently removing code smells contributes to overall code quality, making it easier for developers to collaborate, maintain, and enhance the system over time. In the future, it would be far easier to work on the project as compared to the original code. Example: Code smells like long methods, nested loops were addressed throughout the codebase. For instance, the **checkIn()** and **checkOut()** methods were refactored to reduce complexity and improve code quality.
- **User Experience:** A well-optimized and bug-free hotel management system enhances the user experience for both customers and hotel staff. It ensures smooth booking processes, check-ins, and check-outs, leading to higher customer satisfaction. We have used various checks in the **HotelLogic** class to input integers, Strings, and Booleans. The **viewCurrentBookings()** and **viewCurrentBookingsSpecificCustomer()** methods were improved for better user experience.

Overall, removing these code smells improves the maintainability, readability, performance, and security of the hotel management system, leading to a better user experience and facilitating future development and enhancements. The refactoring process significantly enhanced the Hotel Management System codebase. It is now more robust, efficient, and easier to maintain. The identification and removal of code smells eliminated potential bugs and improved the software's overall reliability. The refactoring efforts contribute to better code quality and lay the foundation for future enhancements and additions to the system. It is important to acknowledge that while the majority of the identified code smells and undesirables were successfully addressed during the refactoring process, some of the remaining issues may be false positives generated by the **DesigniteJava** tool.

# Bibliography


[1] Code smells, Code Smells, https://code-smells.com/.

[2] Refactoring, Refactoring.Guru, https://refactoring.guru/.

[3] Designite. DesigniteJava. https://www.designite-tools.com/designitejava/.

[4] A taxonomy of software smells, https://tusharma.in/smells/IMPL.html.

[5] JetBrains product documentation, https://www.jetbrains.com/help/.

[6] What's the Problem with Long Methods?, Medium, https://medium.com/@joshsaintjacque/whats-the-problem-with-long-methods-72203f516cdb.

[7] Duplicate Code, Refactoring.Guru, https://refactoring.guru/smells/duplicate-code.

[8] SonarSource. Unused loop variable, SonarSource Rules Explorer, https://rules.sonarsource.com/javascript/RSPEC-1481/.

[9] Code Smell: Magic Numbers, Dev.to, https://dev.to/producthackers/code-smell-magic-numbers-3ngc.

[10] Conditional Complexity, Luzkan's Portfolio, https://luzkan.github.io/smells/conditional-complexity.

[11] Linear Search Code Smell, Hacker News, https://news.ycombinator.com/item?id=18988075.

[12] Feature Envy, Refactoring.Guru, https://refactoring.guru/smells/feature-envy.

[13] Primitive Obsession, Refactoring.Guru, https://refactoring.guru/smells/primitive-obsession.

[14] Large Class, Refactoring.Guru, https://refactoring.guru/smells/large-class.

[15] SonarSource. Static mutable fields, SonarSource Rules Explorer, https://rules.sonarsource.com/java/RSPEC-2386/.

[16] Message Chains, Refactoring.Guru, https://refactoring.guru/smells/message-chains.

[17] Shotgun Surgery, Refactoring.Guru, https://refactoring.guru/smells/shotgun-surgery.

[18] Lack of Modularity, Hacker News, https://news.ycombinator.com/item?id=28123854.

[19] SonarSource. Unused import statement, SonarSource Rules Explorer, https://rules.sonarsource.com/java/RSPEC-1128/.

[20] Maximiliano Contieri. Code Smell 13: Empty Constructors, maximilianocontieri.com, https://maximilianocontieri.com/code-smell-13-empty-constructors.

[21] SonarSource. Weak Exception Handling, SonarSource Rules Explorer, https://rules.sonarsource.com/java/RSPEC-112/.

[22] Inefficient Searching, Towards Data Science, https://towardsdatascience.com/7-code-smells-you-should-know-about-and-avoid-b1edf066c3a5.

[23] What is Code Review?, GitLab, https://about.gitlab.com/topics/version-control/what-is-code-review/.