CSE 566 Software Project Process and Quality Management

ASSIGNMENT 5

Kedar Sai Nadh Reddy Kanchi – 1225297164 – kkachi@asu.edu

Part 1

Source Code

For this assignment I have chosen Database Management System that developed during my bachelor's as part of the CSE 207 Object Oriented Programming in Java course.

Overview of Java Database Management System (DBMS)

The Java Database Management System (DBMS) is a sophisticated software application engineered to optimize database interactions specifically for managing user data and roles through a MySQL database interface. Developed using Java programming language and leveraging the robust Java Database Connectivity (JDBC).

Code Functionalities

The Java DBMS is equipped with an array of functionalities that significantly enhance user and role management within organizations. At its core, our system supports comprehensive CRUD (Create, Read, Update, Delete) operations for user management. This functionality allows us as administrators to efficiently add new users, update existing user details, delete users from the system, and retrieve comprehensive lists of user data. These capabilities are complemented by robust role management features that enable the addition of new roles, the assignment of specific roles to users, and the removal of roles, facilitating dynamic and flexible access control.

Further enhancing our capabilities, our system incorporates advanced search functionality that enables us to quickly find specific users by name or email, streamlining user management and improving operational efficiency. We also implement meticulous transaction management protocols to ensure the integrity and consistency of data. We carefully manage each transaction to handle commits and rollbacks effectively, thereby safeguarding against data corruption and ensuring reliable database operations.

Moreover, our system's error handling and connection management frameworks are meticulously designed to handle SQL exceptions and manage database connections securely. This not only enhances the stability and reliability of the system but also ensures that we adhere to best practices in database security and operations management.

Non-Default Metric

I have chosen the M2-Functional Quality Metrics report to get the non-default metrics. This is because the generation of the M2-Functional Quality Metrics report is highly recommended as it facilitates a detailed evaluation of the software's code quality. This analytical tool focuses on essential aspects such as commenting practices, code structure, complexity, and adherence to organizational standards. By systematically identifying deviations from best practices, the report illuminates areas that could negatively impact the maintainability, scalability, and readability of the software. Such insights are invaluable as they guide developers in implementing targeted refactoring strategies to address specific deficiencies in the codebase. Consequently, leveraging this report can significantly enhance software quality, reduce error rates, and simplify the process of future modifications, thereby providing a robust foundation for ongoing software development and maintenance in an academic or professional setting.

Metric Analysis and Interpretation

The Resource Standard Metrics (RSM) tool was employed on the Java based Database Manager System with an emphasis on Complexity and Functional Quality metrics. The Database Manager System has only one java file.

Complexity Metric Analysis and Interpretation

Based on the complexity metrics report generated for the DatabaseManager.java file, we can draw several insights about the quality and design of the software. Here's a detailed interpretation of these metrics:

Overview of Code Metrics

Summary

- Total Lines of Code (LOC): 239
- Effective Lines of Code (eLOC): 207
- Logical Lines of Code (lLOC): 163
- Total Lines including comments and blanks: 253.
- Function Count: 13 functions

Interpretation

The DatabaseManager.java program contains 239 lines of code, indicating a moderate size that should be manageable for a single developer. The effective(eLOC) and logical lines of code are 207 and 163 respectively, which shows a good ratio of lines that directly contribute to functionality versus overall code size. This density suggests that the code is relatively straightforward and not overly padded with inefficiencies or redundancies. The presence of 13 functions indicates that the program has distributed its functionality across multiple, smaller blocks of code, which is beneficial for isolation of responsibilities and easier debugging.

Cyclomatic Complexity Analysis

Summary

- Average Cyclomatic Complexity: 2.31
- Maximum Cyclomatic Complexity: 13 (in `DatabaseManager.main`)

Interpretation

The average cyclomatic complexity**[1]** of 2.31 across the program is indicative of a straightforward logic with a simple control flow with few branching paths, which simplifies understanding and maintenance. The complexity peaks at 13 in the main function, suggesting a concentration of decision-making in this area. This could be a concern for potential errors and may require careful testing and possible refactoring to ensure reliability and maintainability. Lowering this complexity in main could distribute decision points more evenly across other functions or modularize the code better.

Function-Specific Metrics

Summary

- `DatabaseManager.main`: Highest complexity at 13, LOC 58
- Other Functions: Complexity generally 1 or 2, LOC ranges from 10 to 15

The software comprises 13 functions, reflecting a potentially well-structured approach to breaking down the program's functionality.

Interpretation

The main function stands out for its higher complexity and longer code length, which could indicate multiple responsibilities or a central control flow. This suggests that the function might be managing several core operations and could benefit from decomposing into smaller, more focused methods. Whereas the utility functions like the `getConnection`,`executeUpdate` etc, maintain low complexity and shorter lengths pointing to specific, singlepurpose roles which is ideal for unit testing and maintenance. However, the sparse comments across these functions could make it difficult to quickly grasp their intended functionalities and edge cases.

Code Maintainability and Scalability

Summary

- Comment Lines: Only 5 lines throughout the code.
- Comment to Code Ratio: Very Low
- Interface Complexity: Average of 1.85

Interpretation

The scant number of comments in the entire codebase could either indicate that the code is straightforward enough not to require extensive documentation or that there is an area for improvement in terms of code documentation. But still, the low comment-to-code ratio raises concerns regarding the future maintainability of the code. Without sufficient documentation, scaling the software or onboarding new developers could be challenging. Interface complexity being moderate suggests that the method interfaces are not overly complicated, which is positive, but there's room for improvement in documenting these interfaces especially by enhancing comments, particularly in complex sections like the main function, could aid future developers in understanding the business logic faster and facilitate smoother updates and maintenance.

Conclusion of the Complexity Metric Analysis and Interpretation

The use of the Resource Standard Metrics (RSM) tool has provided valuable insights into the Database Management System. The analysis revealed that the DatabaseManager.java program shows a promising code-base structure with generally low complexity across most functions and balanced code distriubution, facilitating easier testing and maintenance. However, the significant complexity in the main function, combined with a lack of sufficient documentation, poses risks to maintainability and understandability. Addressing these issues through targeted refactoring and improved documentation can enhance the overall quality and sustainability of the software.

Non-Default Metric Analysis and Interpretation

The report generated by the Resource Standard Metrics tool provides insights into several aspects of the code quality for the DatabaseManager.java. The key findings focus on function and class metrics, complexity, and coding style issues. Below, I interpret these findings in terms of their implications for the quality and design of the code:

1. **Commenting Practices**: **Insufficient Commenting**: The code lacks adequate comments both at the class level and across various function blocks. Specifically, no preceding comment is present for the class and

multiple functions, which could hinder understandability and maintainability. Most functions lack preceding comments (17.33% quality notices). The overall comments in the code only account for a small fraction of the total lines i.e. 2.1%, which is significantly below the 10% threshold suggesting that the code might not be well documented. This can make the code harder to understand and maintain, especially for new developers or when modifications are needed.

2. **Code Structure: Excessive Line Lengths:** There are numerous instances where the length of the lines exceeds the standard terminal width of 80 characters (22.67% quality notices). Long lines can make the code less readable and complicate code reviews and edits in environments that do not support line wrapping.
3. **Blank Line Usage: Sparse Use of Blank Lines:** The report highlights an underuse of blank lines both within functions and classes (17.33% quality notices). The low percentage of blank lines (less than 10%) might suggest dense coding practices that could affect the readability and ease of navigation through the code.
4. **Exception Handling: Heavy Use of Try-Catch Blocks:** The frequent use of try-catch for exception handling is noted as a potential issue (16.00% quality notices), suggesting that it may be used as a control structure which can lead to "spaghetti code" which makes the logic harder to follow and could lead to hidden bugs.
5. **Function and Class Metrics: High Function Count per File:** With 13 functions defined in a single file, the DatabaseManager class might be doing too much, potentially violating the Single Responsibility Principle.

To address the challenges of maintaining and scaling the `DatabaseManager.java` codebase, it is recommended to enhance the commenting practices by providing comprehensive documentation before each class and function. Refactoring the code to maintain standard line lengths and organizing content with appropriate use of blank lines will further improve readability and structure. Additionally, scrutinizing the use of exception handling to ensure it aligns with best practices for error conditions will help avoid overly complex and convoluted code. Implementing these changes will significantly improve the readability and maintainability of the code, thus enhancing the overall software quality.

Part 2

Introduction to Code Portability

Code portability**[2]** refers to the ability of software to run across different environments, such as operating systems, hardware platforms, or devices, without requiring significant modifications. This characteristic is crucial**[3]** because it enhances the flexibility and scalability of software, enabling developers to reach a broader audience without needing to rewrite or heavily adapt their code for each new platform. Portability is particularly important in today's diverse technological landscape, where applications might need to operate seamlessly on desktops, mobile devices, and cloud-based systems. By designing software with portability in mind, developers can reduce development time and costs, improve software quality, and simplify maintenance and updates, making their applications more competitive and adaptable to evolving technologies.

Literature Review of Metrics for Code Portability

Assessing the portability of code is crucial for developing software that can be easily deployed and executed across a variety of hardware and software platforms. However, quantifying code portability is not a trivial task, as there are no universally accepted standards or metrics. Researchers have proposed several techniques that focus on different aspects of code portability. The main approaches fall into a few broad categories:

1. Measuring the characteristics of the code itself, such as the percentage of platform-independent code, adherence to coding standards, and modular design.
2. Evaluating the performance of the portable code compared to optimized implementations or theoretical performance ceilings.
3. Calculating composite metrics that combine measures of both code characteristics and performance across multiple target platforms.

Basing on this, the key techniques that can be used for measuring code portability are:

1. **Portability Metric:** Although this metric is critical, there isn't a single universal formula to calculate this. Instead, it is about the proportion of these dependencies in comparison to the entire codebase. Therefore, the portability metric measures the percentage of platform-independent code compared to the total code in the application. It is calculated by taking the amount of code that is specific to a particular platform or environment and dividing it by the total amount of code. A higher percentage of platform-independent code indicates better overall portability of the codebase. Overall, we can classify that the degree of the

platform=dependent code is inversely proportional to the portability of the code. Platform dependencies is typically assessed by:

   a. Conducting **static code analysis** using various tools in the market to identify and fix defects, faults, and other issues in the portable code helps ensure it functions properly across different platforms. This includes checking for things like memory leaks, race conditions, and undefined behaviors that could impact portability.
   b. **Interdependency Measurement**: Quantifying the number of interdependencies, or code elements that rely on each other to function properly, can indicate the modularity and reusability of the portable code. Fewer interdependencies generally mean more portable, maintainable code. Static code analysis can be used to identify these relationships between code components. However, the specific details of what counts as "platform specific" can vary widely depending on the context and needs to be defined based on the platforms considered.

2. **Comparing Performance to Optimized Implementations**: This technique**[6]** involves measuring the performance of the portable version of the code and comparing it to a well-recognized, highly optimized implementation of the same application on each target platform. The portable code's performance is then expressed as a fraction or percentage of the optimized implementation's performance. This provides a way to assess the relative performance portability compared to the best-known versions.
   a. The portable code's performance can be increased by following some strict practices like by following some strict **language standard compliance** which evaluates how closely the code adheres to the standard specifications of the programming language used. This can be done by using multiple **compilers** and **setting the warning levels as high** as possible helps ensure the portable code adheres to coding standards and can be compiled successfully across different platforms. Identifying and fixing any compiler warnings helps improve the overall portability of the codebase.
3. **Roofline Performance Mode**l: The roofline performance model**[6]** establishes theoretical performance ceilings or "roofs" for an application based on factors like memory bandwidth and instruction-level parallelism. By analyzing where the portable code's actual performance falls relative to these roofs, you can identify the key hardware characteristics that are limiting its performance portability. The roofline model plots performance as a function of the application's arithmetic intensity (FLOPs per byte of memory access).

Relation of General Code Portability Metrics to the Resource Standard Metrics (RSM) tool

The Resource Standard Metrics (RSM) tool is designed to analyse the compiled source code across the following languags: C, C++, C# and Java. The RSM metrics and analysis capabilities cover a wide range of code elements, including function-level, class-level, and namespace/package-level metrics, which are applicable across the supported programming languages. Therefore, if not directly, the Resource Standard Metrics (RSM) tool**[5]** offers various metrics**[4]** like Cyclomatic Complexity, Lines of Code, and Comment Weight etc which indirectly relate to code portability.

1. **Portability Metric:** This metric focuses on quantifying the proportion of platform-independent code within the codebase. The Macro LOC metric and the White Space Percentage Metric of the RSM tool can inform this analysis.
    a. The size and complexity of macros, as measured by the Macro LOC Metric, can indicate the degree of platform-specific code within the system. Larger, more complex macros are more likely to contain platform-dependent functionality, negatively impacting portability.
    b. Whereas the White Space Percentage Metric measures the amount of whitespace in the code, can be an indirect indicator of code readability and maintainability. A higher white space percentage may suggest a more modular, reusable codebase, which is generally more portable across platforms.
2. **Comparing Performance to Optimized Implementations**: The second portability measure outlined in the prompt involves comparing the performance of the portable code to highly optimized implementations on each target platform. The RSM tool's metrics that can inform this analysis include:
    a. **Function Complexity Metric:** The RSM Function Complexity Metric, which combines cyclomatic complexity and interface complexity, can highlight areas of the code that may be less performant and require further optimization. Lower complexity may indicate simpler control structures, which are typically easier to adapt and modify across different platforms. Thus, higher function complexity is often associated with less portable, more tightly coupled code and a lower combines cyclomatic complexity supports better portability.
    b. **Comment Line and Comment Percent Metric:** The RSM tool's Comment Line and Comment Percent Metric can provide insights into the readability and maintainability of the code. Well-commented, readable code is often easier to optimize for performance across different platforms.

Conclusion

Effective analysis of code portability necessitates a comprehensive evaluation of aspects such as dependencies on platform-specific features, utilization of external libraries, and adherence to programming language standards. While the RSM tool predominantly evaluates metrics related to complexity and maintainability, these metrics can serve as supplementary indicators in portability assessments by identifying areas of code that are overly complex or extensively documented, which may complicate the porting process. Therefore, by leveraging the insights provided by RSM metrics alongside direct evaluations of portability, developers can formulate more effective strategies for creating software that is easily adaptable, requiring minimal modifications to operate across diverse computing environment and compare its performance to the best-known implementations on each target platform.

References

1. Code metrics - Cyclomatic complexity.
2. Software Portability
3. The importance of Software Portability
4. Metrics Definitions
5. What’s Inside Your Source Code?
6. Measuring Performance Portability