Analysis Date: December 5, 2024
Project Name: HMS
SonarQube Version: 24.12.0.100206
Branch: master

Summary of Findings

The static code analysis report identified various issues related to code quality, performance, security, and maintainability. The following is a detailed breakdown of these issues, and specific recommendations for addressing them.

Issues Breakdown

1. Class Variable Fields Should Not Have Public Accessibility

- **Severity**: Major
- **Occurrences**: 82
- **Rule**: Class fields should not be public. They should be private or protected and accessed via getter and setter methods.

2. Anonymous Inner Classes Containing Only One Method Should Become Lambdas

- **Severity**: Minor
- **Occurrences**: 49
- **Rule**: Refactor anonymous inner classes implementing a single method into lambda expressions for better readability and reduced boilerplate.

3. Local Variable and Method Parameter Names Should Comply with a Naming Convention

- **Severity**: Minor
- **Occurrences**: 28
- **Rule**: Local variables and method parameters should follow consistent naming conventions, usually camelCase.

4. Field Names Should Comply with a Naming Convention

- **Severity**: Minor
- **Occurrences**: 23

- **Rule**: Field names should follow consistent naming conventions, usually camelCase for instance variables and uppercase with underscores for constants.

5. Fields in a "Serializable" Class Should Either Be Transient or Serializable

- **Severity**: Major
- **Occurrences**: 19
- **Rule**: Fields in `Serializable` classes should be either marked as `transient` (if they should not be serialized) or should themselves be `Serializable`.

6. Resources Should Be Closed

- **Severity**: Major
- **Occurrences**: 17
- **Rule**: Resources like file streams, database connections, or network connections should be properly closed after use to avoid resource leaks.

7. Package Declaration Should Match Source File Directory

- **Severity**: Minor
- **Occurrences**: 16
- **Rule**: The package declaration in the Java file should match the directory structure where the file is located.

8. Raw Types Should Not Be Used

- **Severity**: Major
- **Occurrences**: 15
- **Rule**: Raw types in generics should be avoided as they bypass type safety, which may lead to runtime errors.

9. Nested Blocks of Code Should Not Be Left Empty

- **Severity**: Minor
- **Occurrences**: 14
- **Rule**: Avoid leaving nested blocks of code empty, as this may indicate incomplete code or bugs.

10. Sections of Code Should Not Be Commented Out

- **Severity**: Minor
- **Occurrences**: 13
- **Rule**: Commented-out code should be removed to reduce clutter and prevent confusion.

11. Unnecessary Imports Should Be Removed

- **Severity**: Minor
- **Occurrences**: 11
- **Rule**: Unused imports should be removed to make the code cleaner and prevent unnecessary dependencies.

12. "Static" Base Class Members Should Not Be Accessed via Derived Types

- **Severity**: Minor
- **Occurrences**: 11
- **Rule**: Static members of the base class should be accessed directly via the base class, not through derived types.

13. Lambdas Should Be Replaced with Method References

- **Severity**: Minor
- **Occurrences**: 11
- **Rule**: Where applicable, replace lambda expressions with method references to improve readability and simplify code.

14. SQL Queries Should Retrieve Only Necessary Fields

- **Severity**: Major
- **Occurrences**: 10
- **Rule**: SQL queries should select only the fields needed, rather than using `SELECT *`, to reduce unnecessary data retrieval and improve performance.

15. Standard Outputs Should Not Be Used Directly to Log Anything

- **Severity**: Minor
- **Occurrences**: 10
- **Rule**: Avoid using `System.out.println` or other standard output methods to log data. Use proper logging frameworks (e.g., `Log4j`, `SLF4J`).

16. String Literals Should Not Be Duplicated

- **Severity**: Minor
- **Occurrences**: 9
- **Rule**: String literals used multiple times should be stored as constants to avoid duplication and make future modifications easier.

17. Unused "Private" Fields Should Be Removed

- **Severity**: Minor
- **Occurrences**: 8
- **Rule**: Remove private fields that are not being used anywhere in the class to reduce clutter and improve maintainability.

18. Package Names Should Comply with a Naming Convention

- **Severity**: Minor
- **Occurrences**: 3
- **Rule**: Package names should follow standard naming conventions (e.g., lowercase letters, domain name reversal).

19. "InterruptedException" and "ThreadDeath" Should Not Be Ignored

- **Severity**: Critical
- **Occurrences**: 3
- **Rule**: `InterruptedException` and `ThreadDeath` should not be ignored. These exceptions must be handled to manage thread states appropriately.

20. Try-Catch Blocks Should Not Be Nested

- **Severity**: Minor
- **Occurrences**: 2
- **Rule**: Avoid nesting try-catch blocks as they complicate error handling and reduce code readability.

21. Unused Local Variables Should Be Removed

- **Severity**: Minor
- **Occurrences**: 2
- **Rule**: Remove local variables that are declared but not used to keep the codebase clean.

22. Local Variables Should Not Shadow Class Fields

- **Severity**: Minor
- **Occurrences**: 2
- **Rule**: Local variables should not shadow class fields, as this can lead to confusion and potential errors.

23. Unused Assignments Should Be Removed

- **Severity**: Minor
- **Occurrences**: 2
- **Rule**: Remove any assignments where the assigned value is never used, as this adds unnecessary clutter to the code.

24. "@Deprecated" Code Should Not Be Used

- **Severity**: Major
- **Occurrences**: 1

- **Rule**: Avoid using deprecated code, as it may be removed in future versions and may not be supported.

25. Track Uses of "TODO" Tags

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Ensure that `TODO` tags are tracked and addressed before deployment to prevent incomplete features or logic from being released.

26. "@Deprecated" Code Marked for Removal Should Never Be Used

- **Severity**: Major
- **Occurrences**: 1
- **Rule**: Deprecated code marked for removal should not be used in the codebase. This avoids potential compatibility issues.

27. Credentials Should Not Be Hard-Coded

- **Severity**: Critical
- **Occurrences**: 1
- **Rule**: Never hard-code credentials (e.g., passwords, API keys) in the code. Use environment variables or secure vaults for managing sensitive data.

28. The Default Unnamed Package Should Not Be Used

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Avoid using the default unnamed package. Always define a proper package name for clarity and maintainability.

29. Mergeable "If" Statements Should Be Combined

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Combine multiple `if` statements that can be logically merged to reduce code complexity and improve readability.

30. Class Names Should Comply with a Naming Convention

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Class names should follow standard conventions, usually PascalCase (e.g., `MyClass`, `StudentRecord`).

31. Multiple Variables Should Not Be Declared on the Same Line

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Avoid declaring multiple variables on the same line to improve clarity. Declare each variable on its own line.

32. Loops Should Not Be Infinite

- **Severity**: Critical
- **Occurrences**: 1
- **Rule**: Ensure that loops have proper exit conditions to avoid infinite loops, which can cause performance issues or application crashes.

33. Cognitive Complexity of Methods Should Not Be Too High

- **Severity**: Major
- **Occurrences**: 1
- **Rule**: Reduce the cognitive complexity of methods to make the code easier to understand and maintain.

34. Ternary Operators Should Not Be Nested

- **Severity**: Minor
- **Occurrences**: 1
- **Rule**: Avoid nesting ternary operators as they can reduce code readability. Use `if`-`else` statements where appropriate.

Recommendations

1. Class Variable Fields Should Not Have Public Accessibility

- **Recommendation**:
  - Change class variables from `public` to `private` or `protected` and provide getter and setter methods for accessing them.

Example:

```
 // Before:
public int employeeId;
// After:
private int employeeId;
public int getEmployeeId() { return employeeId; }
public void setEmployeeId(int employeeId) { this.employeeId = employeeId; }
```

- ○

---

2. Anonymous Inner Classes Containing Only One Method Should Become Lambdas

- **Recommendation**:
  - Refactor anonymous inner classes that implement a single method into lambda expressions.

Example:
```
 // Before:
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) { /* handle action */ }
});
// After:
button.addActionListener(e -> { /* handle action */ });
```

- ○

---

3. Local Variable and Method Parameter Names Should Comply with a Naming Convention

- **Recommendation**:
  - Ensure local variables and method parameters follow `camelCase` naming convention.

Example:
```
 // Before:
int EmployeeAge;
// After:
int employeeAge;
```

- ○

---

4. Field Names Should Comply with a Naming Convention

- **Recommendation**:
  - Make sure fields follow a consistent naming convention, such as `camelCase` for instance variables.

Example:

```
 // Before:
private int EmployeeAge;
// After:
private int employeeAge;
```

- ○

---

5. Fields in a "Serializable" Class Should Either Be Transient or Serializable

- **Recommendation**:
  - Mark non-serializable fields as `transient` or ensure they implement `Serializable`.

Example:

```
 // Before:
private DatabaseConnection connection;
// After:
private transient DatabaseConnection connection;
```

- ○

---

6. Resources Should Be Closed

- **Recommendation**:
  - Always close resources like files, streams, or database connections using try-with-resources or in a `finally` block.

Example:

```
 try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    // Use the resource
} catch (IOException e) {
    // Handle exception
}
```

- ○

---

7. Package Declaration Should Match Source File Directory

- **Recommendation**:
    - Ensure the package declaration matches the file directory structure.

Example:

```
 // If file is in com/example/myapp
package com.example.myapp;
```

    ○

---

8. Raw Types Should Not Be Used

- **Recommendation**:
    - Avoid using raw types for generics. Always specify the type parameters.

Example:

```
 // Before:
List list = new ArrayList();
// After:
List<String> list = new ArrayList<>();
```

    ○

---

9. Nested Blocks of Code Should Not Be Left Empty

- **Recommendation**:
    - Avoid leaving empty blocks of code. If not necessary, remove the block entirely.

Example:

```
 // Before:
if (condition) { }
// After:
if (condition) {
    // Add logic or remove block
}
```

    ○

---

10. Sections of Code Should Not Be Commented Out

- **Recommendation**:

    - Remove commented-out code or ensure it is properly documented if left for later.

Example:
 // Before:
// int x = 0;
// After:
// Remove commented code or finalize it.

    -

---

11. Unnecessary Imports Should Be Removed

- **Recommendation**:
    - Remove unused imports to clean up the code and avoid potential confusion.

Example:
 // Before:
import java.util.List;
import java.util.ArrayList;
// After:
import java.util.ArrayList;

    -

---

12. "Static" Base Class Members Should Not Be Accessed via Derived Types

- **Recommendation**:
    - Access static members of base classes directly via the class name, not through instances or derived types.

Example:
 // Before:
DerivedClass.staticMethod();
// After:
BaseClass.staticMethod();

    -

---

13. Lambdas Should Be Replaced with Method References

- **Recommendation**:
    - Refactor lambdas into method references where appropriate.

Example:
 // Before:
list.forEach(x -> System.out.println(x));
// After:
list.forEach(System.out::println);

-

---

14. SQL Queries Should Retrieve Only Necessary Fields

- **Recommendation**:
    - Avoid `SELECT *` in SQL queries; always select only the columns that are needed.

Example:
 // Before:
SELECT * FROM users;
// After:
SELECT username, email FROM users;

-

---

15. Standard Outputs Should Not Be Used Directly to Log Anything

- **Recommendation**:
    - Use a logging framework (e.g., Log4j, SLF4J) instead of `System.out.println`.

Example:
 // Before:
System.out.println("Error message");
// After:
Logger logger = LoggerFactory.getLogger(MyClass.class);
logger.error("Error message");

-

---

16. String Literals Should Not Be Duplicated

- **Recommendation**:
  - Avoid duplicating string literals. Use constants instead.

Example:

```
 // Before:
String error = "File not found";
// After:
public static final String ERROR_FILE_NOT_FOUND = "File not found";
```

-

---

17. Unused "Private" Fields Should Be Removed

- **Recommendation**:
  - Remove unused private fields to keep the code clean.

Example:

```
 // Before:
private int unusedField;
// After:
// Remove unusedField
```

-

---

18. Package Names Should Comply with a Naming Convention

- **Recommendation**:
  - Use lowercase and reverse domain name conventions for package names.

Example:

```
 // Before:
package com.MyApp;
// After:
package com.myapp;
```

-

---

19. "InterruptedException" and "ThreadDeath" Should Not Be Ignored

- **Recommendation**:
    - Properly handle `InterruptedException` by either restoring the thread's interrupt status or terminating appropriately.

Example:
// Before:
try { Thread.sleep(1000); } catch (InterruptedException e) { }
// After:
try { Thread.sleep(1000); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }

    -

---

20. Try-Catch Blocks Should Not Be Nested

- **Recommendation**:
    - Avoid nesting try-catch blocks. Flatten them if possible to improve readability.

Example:
```
 // Before:
try {
   try { /* some code */ } catch (IOException e) { /* handle IOException */ }
} catch (Exception e) { /* handle Exception */ }
// After:
try { /* some code */ } catch (IOException e) { /* handle IOException */ } catch (Exception e) { /*
handle Exception */ }
```

    -

---

21. Unused Local Variables Should Be Removed

- **Recommendation**:
    - Remove local variables that are declared but never used in the method.

Example:
 // Before:
int unusedVariable = 5;
// After:
// Remove unused variable

    -

---

22. Local Variables Should Not Shadow Class Fields

- **Recommendation**:
  - Avoid naming local variables the same as class fields to prevent confusion.

Example:
```
 // Before:
private int value;
public void setValue(int value) {
   this.value = value; // Shadowing class field
}
// After:
public void setValue(int newValue) {
   this.value = newValue; // No shadowing
}
```

  -

---

23. Unused Assignments Should Be Removed

- **Recommendation**:
  - Remove assignments to variables that are never used.

Example:
```
 // Before:
int temp = 10;
// After:
// Remove unused assignments
```

  -

---

24. "@Deprecated" Code Should Not Be Used

- **Recommendation**:
  - Avoid using deprecated code. Look for updated or alternative methods.
  - **Example**:

java // Before: @Deprecated public void oldMethod() { } // After: // Use the recommended alternative method ```

---

25. Track Uses of "TODO" Tags

- **Recommendation**:
    - Remove or complete `TODO` comments in the code.

Example:
 // Before:
// TODO: Implement logging
// After:
// Implemented logging functionality

    -

---

26. Credentials Should Not Be Hard-Coded

- **Recommendation**:
    - Use environment variables or secure vaults for credentials instead of hard-coding them.

Example:
 // Before:
String password = "admin123"; // Hardcoded password
// After:
String password = System.getenv("DB_PASSWORD");

    -

---

27. The Default Unnamed Package Should Not Be Used

- **Recommendation**:
    - Always use named packages and avoid using the default unnamed package.

Example:
 // Before:
// No package declared
// After:
package com.example.project;

    -

---

28. Mergeable "if" Statements Should Be Combined

- **Recommendation**:
  - Combine multiple `if` statements with the same condition into one.

Example:

```
 // Before:
if (a == 1) { }
if (a == 1) { }
// After:
if (a == 1) { }
```

  -

---

29. Ternary Operators Should Not Be Nested

- **Recommendation**:
  - Avoid nesting ternary operators. Use `if-else` for better readability.

Example:

```
 // Before:
result = (x > 0) ? (y > 0 ? "Positive" : "Negative") : "Zero";
// After:
if (x > 0) {
    result = (y > 0) ? "Positive" : "Negative";
} else {
    result = "Zero";
}
```

  -

---

30. Class Names Should Comply with a Naming Convention

- **Recommendation**:
  - Ensure class names are in PascalCase and follow conventions.

Example:

```
 // Before:
class employee { }
// After:
class Employee { }
```

-

---

31. Multiple Variables Should Not Be Declared on the Same Line

- **Recommendation**:
    - Declare one variable per line for clarity.

Example:
 // Before:
int x = 0, y = 1;
// After:
int x = 0;
int y = 1;

-

---

32. Loops Should Not Be Infinite

- **Recommendation**:
    - Ensure loops have a clear termination condition to prevent infinite loops.

Example:
 // Before:
while (true) { }
// After:
while (x < 10) { }

-

---

33. Cognitive Complexity of Methods Should Not Be Too High

- **Recommendation**:
    - Refactor methods with high cognitive complexity into smaller, more manageable pieces.

Example:
 // Before:
public void complexMethod() { /* complex code */ }
// After:
public void simpleMethod() { /* simple code */ }

◦

---

34. Mergeable "if" Statements Should Be Combined

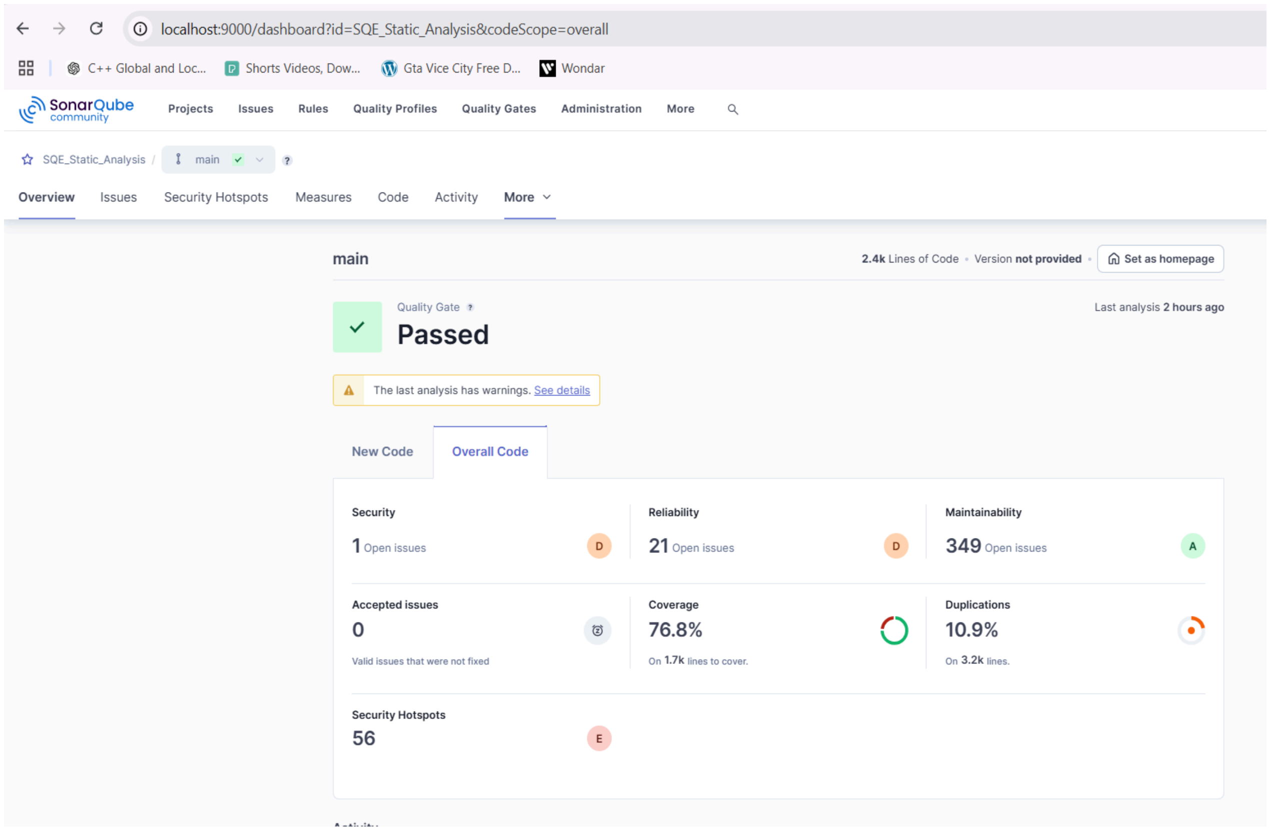
- **Recommendation**:
    - Combine multiple `if` statements with the same condition into one to reduce redundancy.

Example:

```
 // Before:
if (x == 1) { }
if (x == 1) { }
// After:
if (x == 1) { }
```

◦

SonarQube Analysis


localhost:9000/dashboard?id=SQE_Static_Analysis&codeScope=overall

main — 2.4k Lines of Code · Version not provided

Quality Gate: Passed — Last analysis 2 hours ago

The last analysis has warnings. See details

Security	Reliability	Maintainability
1 Open issues (D)	21 Open issues (D)	349 Open issues (A)

Accepted issues	Coverage	Duplications
0 — Valid issues that were not fixed	76.8% — On 1.7k lines to cover.	10.9% — On 3.2k lines.

Security Hotspots: 56 (E)

Command used to run SonarQube on Local Server

```
sonar-scanner -D"sonar.projectKey=SQE_Static_Analysis"
-D"sonar.sources=C:\Users\92309\Downloads\Hotel-Management-System-master\Hotel-Management-System-master\Hotel Management System\src\hotel\management\system"
-D"sonar.tests=C:\Users\92309\Downloads\Hotel-Management-System-master\Hotel-Management-System-master\Hotel Management System\src\hotel\management\tests"
-D"sonar.java.binaries=C:\Users\92309\Downloads\Hotel-Management-System-master\Hotel-Management-System-master\out\production\Hotel-Management-System-master\hotel\management\system"
-D"sonar.coverage.jacoco.xmlReportPaths=C:\Users\92309\Downloads\Hotel-Management-System-master\Hotel-Management-System-master\jacoco-report.xml"
-D"sonar.host.url=http://localhost:9000"
-D"sonar.token=sqp_b2609a363663b563935e4325ac39096d176a4242"
```

- *It is run in the directory of project through cmd via Sonar Scan to run on Local Host Server 9000*