1. Identified Code Smell: Duplicate Code

Class: StatisticsChartRenderer

Methods: getDefinition(), getDef()

The StatisticsChartRenderer class in the Mango system contained two methods, getDefinition() and getDef(), both of which returned the same constant value (definition). This redundancy violates the DRY (Don't Repeat Yourself) principle, making maintenance more challenging as any changes to the logic must be applied in multiple places. The presence of duplicate methods adds unnecessary complexity and reduces readability.

Method: getDefinition(), getDef()

The getDefinition() and getDef() methods both return a constant definition value, leading to unnecessary duplication and reducing the clarity of the StatisticsChartRenderer class.

Explain why the flagged class/method is smelly (be specific).

The presence of duplicate methods in the StatisticsChartRenderer class leads to several issues:

- **Violation of DRY Principle:** Code repetition increases maintenance overhead. If one method is modified but the other is not, inconsistencies may arise.
- **Increased Maintenance Effort:** Having identical implementations in multiple places means that any change must be applied to both methods, making future updates more complex.
- **Code Clutter:** Unnecessary duplication adds extra lines of code, reducing overall readability and maintainability.

Do you agree that the detected smell is an actual smell? Justify your answer.

Yes, this is a valid code smell because having two identical methods serves no functional purpose. It introduces redundancy, making the code harder to maintain. Refactoring these methods ensures better maintainability, improves code readability, and prevents potential inconsistencies.

List and describe in detail the refactoring (i.e., the code changes) used to remove the smell.

Refactoring Strategy: Apply Method Delegation

Refactoring Steps

- **Step 1**: Identify that both getDefinition() and getDef() return the same value.
- **Step 2**: Keep getDefinition() as the primary method.
- **Step 3**: Modify getDef() to delegate its logic to getDefinition(), removing duplicate code.

This approach preserves API compatibility while removing unnecessary redundancy.

Refactored Code

Before Refactoring:

```
public static ImplDefinition getDefinition() {
    return definition;
}

public ImplDefinition getDef() {
    return definition;
}
```

After Refactoring:

```
public static ImplDefinition getDefinition() {
    return definition;
}

public ImplDefinition getDef() {
    return getDefinition(); // Delegating to getDefinition()
}
```

Method Delegation ensures that the logic exists in only one place.

- If getDefinition() changes in the future, getDef() will automatically reflect those changes.
- This approach avoids unnecessary method duplication while preserving API compatibility.

Verification & Testing

- **Ran unit tests:** Ensured that both getDefinition() and getDef() return the correct value.
- **Re-ran code smell detection tool:** Confirmed that the duplicate code smell was removed.
- **Checked for new code smells:** No new issues were introduced by the refactoring.

Summary

This refactoring successfully eliminated duplicate code, reducing maintenance effort and improving the structure of StatisticsChartRenderer. Using method delegation ensures that future changes to getDefinition() automatically propagate to getDef(), enhancing code maintainability and clarity.

2. Identified Code Smell: Long Method (Nested Try Blocks)

Class: DataSourceEditDwr.java

Method: galilTestCommand()

The galilTestCommand() method in the DataSourceEditDwr.java file contained deeply nested try-catch blocks, making the code difficult to read and maintain. The method was responsible for both executing a command and handling errors, violating the Single Responsibility Principle (SRP).

Explain why the flagged class/method is smelly (be specific).

- **Deep Nesting:** The method contained multiple try-catch blocks nested within each other, reducing readability and making debugging difficult.
- **Violation of SRP:** The method handled both execution and error handling, making it responsible for multiple tasks rather than a single well-defined responsibility.
- **Low Maintainability:** Any modification required developers to navigate through multiple levels of nested structures, making changes risky and complex.

Do you agree that the detected smell is an actual smell? Justify your answer.

Yes, this is an actual code smell. The deeply nested structure complicates readability, and separating concerns improves modularity and maintainability. Extracting error-handling logic into a separate method allows better testing, debugging, and future modifications.

List and describe in detail the refactoring (i.e., the code changes) used to remove the smell.

Refactoring Strategy: Extract Method

Steps Taken:

1. Created a new method executeGalilCommand() to handle command execution separately.
2. Moved the inner try block into executeGalilCommand(), reducing nesting in the original method.
3. The main galilTestCommand() method now only deals with high-level error handling and permissions.

This refactoring improves readability, separates concerns, and enhances maintainability.

Refactored Code

Before Refactoring:

```
@MethodFilter
public String galilTestCommand(String host, int port, int timeout, String command) {
    User user = Common.getUser();
    Permissions.ensureDataSourcePermission(user);

    try {
        GalilCommandTester tester = new GalilCommandTester(getResourceBundle(), host, port, timeout, command);
        try {
            tester.join();
            return tester.getResult();
        }
        catch (InterruptedException e) {
            return e.getMessage();
        }
    }
    catch (IOException e) {
        return e.getMessage();
    }
}
```

After Refactoring:

```
@MethodFilter
public String galilTestCommand(String host, int port, int timeout, String command) {
    User user = Common.getUser();
    Permissions.ensureDataSourcePermission(user);

    try {
        return executeGalilCommand(host, port, timeout, command);
    } catch (IOException e) {
        return e.getMessage();
    }
}

private String executeGalilCommand(String host, int port, int timeout, String command) throws IOException {
    GalilCommandTester tester = new GalilCommandTester(getResourceBundle(), host, port, timeout, command);
    try {
        tester.join();
        return tester.getResult();
    } catch (InterruptedException e) {
        return e.getMessage();
    }
}
```

Give the rationale of the chosen refactoring operations.

- **Improves readability:** Eliminating deeply nested try blocks makes the method easier to follow.
- **Enhances maintainability:** Separating execution logic from error handling improves modularity.
- **Reduces complexity:** Simplifying the structure makes future modifications and debugging more manageable.

Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support (or tool).

- Manually extracted the command execution logic into executeGalilCommand().
- Ensured that all error handling remains in the main method while delegating execution to the new method.

Verification & Testing

- **Ran unit tests:** Confirmed that galilTestCommand() and executeGalilCommand() function correctly.
- **Re-ran code smell detection tool:** Verified that the Long Method smell was removed.
- **Checked for new code smells:** Ensured that no new issues were introduced due to refactoring.

Summary

This refactoring successfully improved readability and maintainability by reducing deep nesting in the method. Separating concerns between execution and error handling makes the code easier to understand, debug, and modify. These changes enhance the overall structure of DataSourceEditDwr.j

3. Identified Code Smell: Unoptimized Iteration (Performance Smell)

Class: Network.java

Method: addressPathsToString()

The addressPathsToString() method in the Network.java file contained an inefficient iteration over keySet(), retrieving values using pathsByAddress.get(address). This resulted in unnecessary lookup operations, increasing computational complexity.

Explain why the flagged class/method is smelly (be specific).

- **Redundant Lookups:** The method first iterates over keySet() and then retrieves values using a separate lookup, causing performance inefficiencies.
- **Increased Computational Complexity:** Unnecessary O(n) operations are performed, leading to suboptimal performance.
- **Readability and Efficiency:** Using a more direct approach improves clarity and performance without altering the method's intended functionality.

Do you agree that the detected smell is an actual smell? Justify your answer.

Yes, this is a definite code smell because the use of keySet() iteration followed by explicit get() calls introduces unnecessary complexity and inefficiencies in the code. The redundant lookups increase time complexity, making each lookup an additional operation that could have been avoided by iterating over entrySet() directly. This inefficiency becomes particularly noticeable in large datasets, where repeated lookups add significant performance overhead.

Furthermore, maintaining the original structure not only affects performance but also reduces readability. The get() calls obscure the simplicity of direct key-value access, making the method harder to follow and increasing the chances of future errors if modifications are required. By using entrySet(), we eliminate these redundant operations, making the code cleaner, more efficient, and easier to maintain.

Additionally, adhering to best coding practices suggests that direct key-value access should always be preferred when both are needed within a loop. The refactoring enhances correctness while ensuring that future iterations remain scalable and optimized for performance.

List and describe in detail the refactoring (i.e., the code changes) used to remove the smell.

Refactoring Strategy: Replace keySet() Iteration with entrySet() Iteration

Steps Taken:

1. Instead of iterating over keySet() and performing redundant lookups, switched to directly iterating over entrySet().
2. Accessed values directly within the loop, eliminating unnecessary get() calls.
3. Verified that the method's behavior remains unchanged after refactoring.

This refactoring improves performance by reducing lookup overhead, simplifies the iteration logic, and enhances code maintainability.

Refactored Code

Before Refactoring:

```
public String addressPathsToString() {
    StringBuilder sb = new StringBuilder();
    sb.append(c:'[');
    boolean first = true;
    for (Long address : pathsByAddress.keySet()) {
        if (first)
            first = false;
        else
            sb.append(str:", ");
        sb.append(pathsByAddress.get(address));
        sb.append(Address.toString(address));
    }
    sb.append(c:']');
    return sb.toString();
}
```

After Refactoring:

```
public String addressPathsToString() {
    StringBuilder sb = new StringBuilder();
    sb.append(c:'[');
    boolean first = true;

    for (Map.Entry<Long, NetworkPath> entry : pathsByAddress.entrySet()) {
        if (first)
            first = false;
        else
            sb.append(str:", ");

        sb.append(entry.getValue()); // Directly accessing the NetworkPath value
        sb.append(Address.toString(entry.getKey())); // Accessing the key
    }

    sb.append(c:']');
    return sb.toString();
}
```

Give the rationale of the chosen refactoring operations.

- **Eliminates redundant lookups:**The keySet() iteration required additional get() calls to retrieve values, causing unnecessary computational overhead. By directly using entrySet(), we eliminate these redundant lookups, improving efficiency and reducing execution time.
- **Improves performance:** This refactoring significantly reduces the number of operations performed when accessing map entries. In large datasets, the previous approach resulted in repetitive key-value retrieval, whereas entrySet() optimizes the process by providing direct key-value access in a single iteration.
- **Simplifies code:** Reduces unnecessary complexity while preserving functionality. A cleaner iteration approach improves readability, making it easier for developers to understand and maintain.

Explain what code changes you had to do manually (if any), in addition to the changes performed with the IDE's support (or tool).

- **Manually Updated Loop Structure:** The initial implementation iterated over keySet() and then accessed values using get(). We manually replaced this by iterating directly over entrySet(), reducing unnecessary lookups and improving performance.

- **Ensured Correct Key-Value Access:** Since entrySet() provides both key and value in one iteration, we adjusted variable assignments to correctly handle both within the loop, ensuring proper functionality.

Verification & Testing

- **Ran unit tests:** Confirmed that addressPathsToString() still returns the expected output.
- **Re-ran code smell detection tool:** Verified that the Unoptimized Iteration smell was removed.
- **Checked for new code smells:** Ensured that no new issues were introduced due to refactoring.

Summary

This refactoring successfully removed the performance bottleneck by eliminating redundant lookups. The improved iteration approach enhances efficiency while maintaining code clarity and correctness. These changes contribute to better maintainability and performance of the Network.java class.