Using jSparrow: Our Experience and Insights

College of Engineering and Architecture
Al-Yamamah University
Kingdom of Saudi Arabia, Riyadh

SWE 402: Software Maintenance and Evolution

Prepared By:

YAZED ALKHALAF - 202211123
SAIMAN TAKLAS - 202021400
ALI BA WAZIR - 202211018

Submitted To:

Dr. Khaled Makhadmeh

May 6, 2025

1 Introduction

jSparrow is a refactoring tool for Java that automates the identification and resolution of code smells, aiming to improve the structure, performance, and maintainability of software. In this project, we document our experience using the free version of jSparrow to refactor a sample Java codebase of approximately 100 lines. The objective was to test how well the tool handles various common bad smells that were deliberately injected into our code.

2 The Test Subject

We created a basic student management system containing multiple common Java bad smells such as redundant semicolons, improper use of string equality, unused imports, and poor enum handling. The original version of the test code is shown in **Listing 1**.

Listing 1: Original Test Subject Java Code

```java
package main;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.Random; // unused
import java.util.Hashtable; // unused

public class StudentManagementSystem {
    ArrayList<String> studentNames = new ArrayList<>();
        ArrayList<Integer> studentIds = new ArrayList<>();
        ArrayList<Double> studentGPAs = new ArrayList<>();;

    public void menu() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("1. Add");
            System.out.println("2. Show");
            System.out.println("3. Find");
            System.out.println("4. Exit");
            int opt = scanner.nextInt(); scanner.nextLine();
            if (opt == 1) {
                add(scanner);
            } else {
                if (opt == 2) {
                    show();
                } else {
                    if (opt == 3) {
                        find(scanner);
                } else {
                    if (opt == 4) {
                        System.out.println("Bye"); break;
                    } else {
                        System.out.println("Wrong");
                    }
                }
            }
        }
    }
    scanner.close();
}

void add(Scanner scanner) {
    System.out.println("Name?");
    String name = scanner.nextLine();
    System.out.println("ID?");
    int id = scanner.nextInt(); scanner.nextLine();
    System.out.println("GPA?");
    double gpa = scanner.nextDouble(); scanner.nextLine();

    studentNames.add(name);
    studentIds.add(id);
    studentGPAs.add(gpa);

    System.out.println("Added");
}

void show() {
    for (int i = 0; i < studentNames.size(); i++) {
        if (!(studentNames.get(i) == null)) {
            if (studentNames.get(i).toString().equals("")) {
                System.out.println("Empty name");
            } else {
                System.out.println("Name: " + studentNames.get(i));
            }
        }
        System.out.println("ID: " + studentIds.get(i));
        System.out.println("GPA: " + studentGPAs.get(i));;
    }
}

void find(Scanner scanner) throws IllegalArgumentException {
    System.out.println("Search name:");
    String q = scanner.nextLine();
    if (q == null || q.equals("")) {
        throw new IllegalArgumentException("Bad input");
    }
```

```
        boolean found = false;
        for (int i = 0; i < studentNames.size(); i++) {
            if (studentNames.get(i).equalsIgnoreCase(q)) {
                System.out.println("Found: " + studentNames.get(i) + ",
                    ID: " + studentIds.get(i));
                found = true;
                break;
            }
        }

        if (!found) {
            System.out.println("Not found");
        }
    }

    void deadMethod1() {
        String a = "useless";
        String b = "code";
        System.out.println(a + b); // never called
    }

    void messyEnums(Color c) {
        if (c.equals(Color.RED)) {
            System.out.println("Red");
        } else if (c.equals(Color.GREEN)) {
            System.out.println("Green");
        } else if (c.equals(Color.BLUE)) {
            System.out.println("Blue");
        }
    }

    public static void main(String[] args) {
        StudentManagementSystem s = new StudentManagementSystem();
            s.menu();
    }

    enum Color {
        RED, GREEN, BLUE
    }
}
```

3 Refactored Code by jSparrow

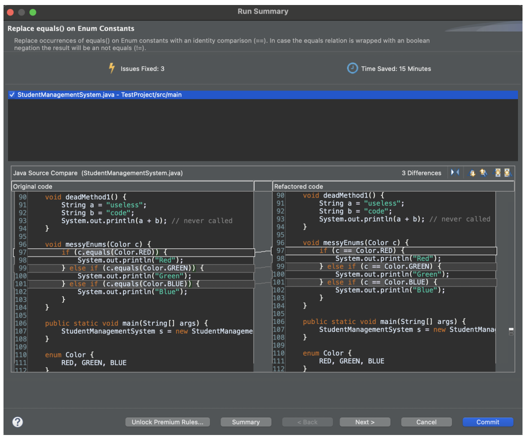
After running jSparrow, we obtained the refactored version of the same program. The transformed version is shown in **Listing 2**.

Listing 2: Refactored Code Generated by jSparrow

```java
package main;

import java.util.ArrayList;
import java.util.Scanner;

public class StudentManagementSystem {
    ArrayList<String> studentNames = new ArrayList<>();
        ArrayList<Integer> studentIds = new ArrayList<>();
        ArrayList<Double> studentGPAs = new ArrayList<>();;

    public void menu() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("1. Add");
            System.out.println("2. Show");
            System.out.println("3. Find");
            System.out.println("4. Exit");
            int opt = scanner.nextInt(); scanner.nextLine();
            if (opt == 1) {
                add(scanner);
            } else {
                if (opt == 2) {
                    show();
                } else {
                    if (opt == 3) {
                        find(scanner);
                    } else {
                        if (opt == 4) {
                            System.out.println("Bye"); break;
                        } else {
                            System.out.println("Wrong");
                        }
                    }
                }
            }
        }
        scanner.close();
    }

    void add(Scanner scanner) {
        System.out.println("Name?");
        String name = scanner.nextLine();
        System.out.println("ID?");
        int id = scanner.nextInt(); scanner.nextLine();
        System.out.println("GPA?");
        double gpa = scanner.nextDouble(); scanner.nextLine();

        studentNames.add(name);
        studentIds.add(id);
        studentGPAs.add(gpa);

        System.out.println("Added");
    }

    void show() {
        for (int i = 0; i < studentNames.size(); i++) {
            if (!(studentNames.get(i) == null)) {
                if ("".equals(studentNames.get(i))) {
                    System.out.println("Empty name");
                } else {
                    System.out.println("Name: " + studentNames.get(i));
                }
            }
            System.out.println("ID: " + studentIds.get(i));
            System.out.println("GPA: " + studentGPAs.get(i));
        }
    }

    void find(Scanner scanner) {
        System.out.println("Search name:");
        String q = scanner.nextLine();
        if (q == null || "".equals(q)) {
            throw new IllegalArgumentException("Bad input");
        }

        boolean found = false;
        for (int i = 0; i < studentNames.size(); i++) {
            if (studentNames.get(i).equalsIgnoreCase(q)) {
                System.out.println("Found: " + studentNames.get(i) + ",
                    ID: " + studentIds.get(i));
                found = true;
                break;
            }
        }

        if (!found) {
            System.out.println("Not found");
        }
    }

    void deadMethod1() {
        String a = "useless";
        String b = "code";
        System.out.println(a + b); // never called
    }

    void messyEnums(Color c) {
        if (c == Color.RED) {
            System.out.println("Red");
        } else if (c == Color.GREEN) {
            System.out.println("Green");
        } else if (c == Color.BLUE) {
            System.out.println("Blue");
        }
    }

    public static void main(String[] args) {
        StudentManagementSystem s = new StudentManagementSystem();
            s.menu();
    }

    enum Color {
        RED, GREEN, BLUE
    }
}
```

4 Analysis of Refactoring Rules

In this section, we analyze the impact of each rule applied by jSparrow using its visual interface. The images highlight the original code on the left and the refactored version on the right.



Replace equals() on Enum Constants

Replace occurrences of equals() on Enum constants with an identity comparison (==). In case the equals relation is wrapped with an boolean negation the result will be an not equals (!=).

Issues Fixed: 3

Time Saved: 15 Minutes

✓ StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java) 3 Differences

Original code

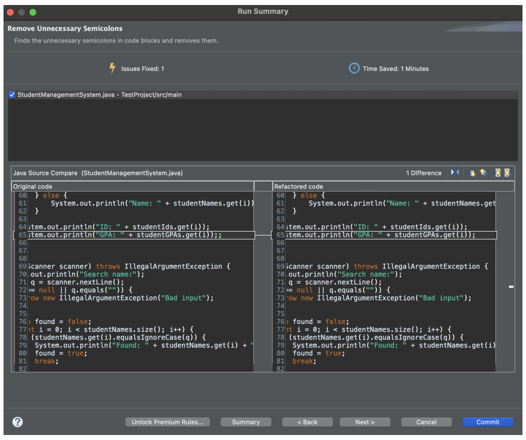
```
    void deadMethod1() {
        String a = "useless";
        String b = "code";
        System.out.println(a + b); // never called
    }

    void messyEnums(Color c) {
        if (c.equals(Color.RED)) {
            System.out.println("Red");
        } else if (c.equals(Color.GREEN)) {
            System.out.println("Green");
        } else if (c.equals(Color.BLUE)) {
            System.out.println("Blue");
        }
    }

    public static void main(String[] args) {
        StudentManagementSystem s = new StudentManageme
    }

    enum Color {
        RED, GREEN, BLUE
    }
```

Refactored code

```
    void deadMethod1() {
        String a = "useless";
        String b = "code";
        System.out.println(a + b); // never called
    }

    void messyEnums(Color c) {
        if (c == Color.RED) {
            System.out.println("Red");
        } else if (c == Color.GREEN) {
            System.out.println("Green");
        } else if (c == Color.BLUE) {
            System.out.println("Blue");
        }
    }

    public static void main(String[] args) {
        StudentManagementSystem s = new StudentManag
    }

    enum Color {
        RED, GREEN, BLUE
    }
```

Figure 1: Replacement of equals() with == for Enum Constants

Rule: Replace equals() on Enum Constants. This transformation replaced the use of `.equals()` with `==` when comparing enum constants. Since enums are singleton instances in Java, using `==` is both safe and more performant. This change improves both clarity and efficiency.



Remove Unnecessary Semicolons

Finds the unnecessary semicolons in code blocks and removes them.

☑ StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java)

Original code

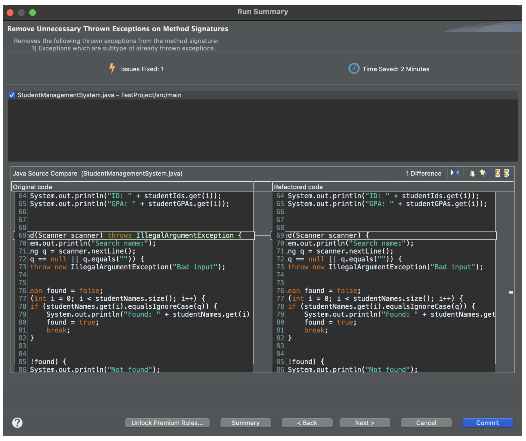
```
 } else {
     System.out.println("Name: " + studentNames.get(i))
 }

;tem.out.println("ID: " + studentIds.get(i));
;tem.out.println("GPA: " + studentGPAs.get(i));;



;canner scanner) throws IllegalArgumentException {
.out.println("Search name:");
 q = scanner.nextLine();
== null || q.equals("")) {
'ow new IllegalArgumentException("Bad input");


ı found = false;
ıt i = 0; i < studentNames.size(); i++) {
 (studentNames.get(i).equalsIgnoreCase(q)) {
 System.out.println("Found: " + studentNames.get(i) + "
 found = true;
 break;
```

Refactored code

```
 } else {
     System.out.println("Name: " + studentNames.get
 }

;tem.out.println("ID: " + studentIds.get(i));
;tem.out.println("GPA: " + studentGPAs.get(i));



;canner scanner) throws IllegalArgumentException {
.out.println("Search name:");
 q = scanner.nextLine();
== null || q.equals("")) {
'ow new IllegalArgumentException("Bad input");


ı found = false;
ıt i = 0; i < studentNames.size(); i++) {
 (studentNames.get(i).equalsIgnoreCase(q)) {
 System.out.println("Found: " + studentNames.get(i)
 found = true;
 break;
```

Figure 2: Remove Unnecessary Semicolons

Rule: Remove Unnecessary Semicolons. In this figure, jSparrow successfully removed a dangling semicolon after a method call. However, the tool failed to detect multiple semicolons on the same line in this statement: `ArrayList<String> studentNames = new ArrayList<>(); ArrayList<Integer> studentIds = new ArrayList<>(); ArrayList<Double> studentGPAs = new ArrayList<>();;` — showing a limitation in multi-statement detection.



Remove Unnecessary Thrown Exceptions on Method Signatures

Removes the following thrown exceptions from the method signature:
1) Exceptions which are subtype of already thrown exceptions.

☑ StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java)

Original code

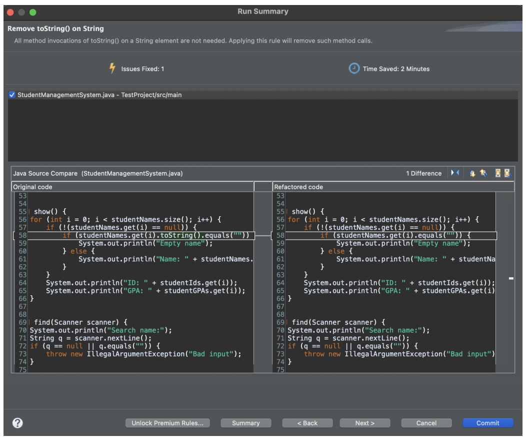
```
System.out.println("ID: " + studentIds.get(i));
System.out.println("GPA: " + studentGPAs.get(i));



ıd(Scanner scanner) throws IllegalArgumentException {
:em.out.println("Search name:");
.ng q = scanner.nextLine();
q == null || q.equals("")) {
throw new IllegalArgumentException("Bad input");


.ean found = false;
(int i = 0; i < studentNames.size(); i++) {
if (studentNames.get(i).equalsIgnoreCase(q)) {
    System.out.println("Found: " + studentNames.get(i)
    found = true;
    break;
}


!found) {
System.out.println("Not found");
```

Refactored code

```
System.out.println("ID: " + studentIds.get(i));
System.out.println("GPA: " + studentGPAs.get(i));



ıd(Scanner scanner) {
:em.out.println("Search name:");
.ng q = scanner.nextLine();
q == null || q.equals("")) {
throw new IllegalArgumentException("Bad input");


.ean found = false;
(int i = 0; i < studentNames.size(); i++) {
if (studentNames.get(i).equalsIgnoreCase(q)) {
    System.out.println("Found: " + studentNames.get
    found = true;
    break;
}


!found) {
System.out.println("Not found");
```

Figure 3: Remove Unnecessary Thrown Exceptions

Rule: Remove Unnecessary Thrown Exceptions on Method Signatures. The `throws IllegalArgumentException` clause was removed from the method signature. This is safe and improves cleanliness because the exception is unchecked and does not need to be declared. The method behavior remains unchanged.



Remove toString() on String

All method invocations of toString() on a String element are not needed. Applying this rule will remove such method calls.

Issues Fixed: 1

Time Saved: 2 Minutes

StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java)

1 Difference

Original code

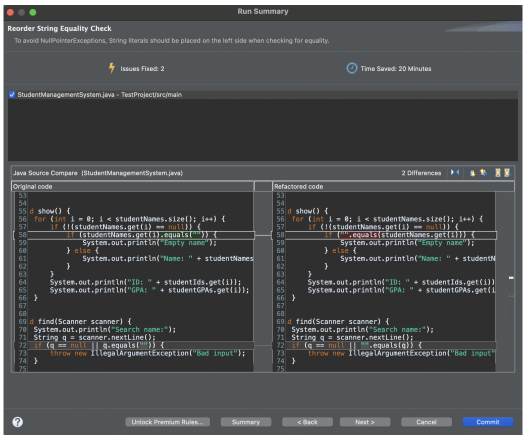
```
 show() {
for (int i = 0; i < studentNames.size(); i++) {
    if (!(studentNames.get(i) == null)) {
        if (studentNames.get(i).toString().equals(""))
            System.out.println("Empty name");
        } else {
            System.out.println("Name: " + studentNames.
        }
    }
    System.out.println("ID: " + studentIds.get(i));
    System.out.println("GPA: " + studentGPAs.get(i));
}


 find(Scanner scanner) {
System.out.println("Search name:");
String q = scanner.nextLine();
if (q == null || q.equals("")) {
    throw new IllegalArgumentException("Bad input");
}
```

Refactored code

```
 show() {
for (int i = 0; i < studentNames.size(); i++) {
    if (!(studentNames.get(i) == null)) {
        if (studentNames.get(i).equals("")) {
            System.out.println("Empty name");
        } else {
            System.out.println("Name: " + studentNa
        }
    }
    System.out.println("ID: " + studentIds.get(i));
    System.out.println("GPA: " + studentGPAs.get(i)
}


 find(Scanner scanner) {
System.out.println("Search name:");
String q = scanner.nextLine();
if (q == null || q.equals("")) {
    throw new IllegalArgumentException("Bad input")
}
```

Figure 4: Remove toString() on String

Rule: Remove toString() on String. The tool identified a redundant `toString()` call on a String instance. Since `studentNames.get(i)` already returns a String, the call was unnecessary. Removing it avoids clutter and improves clarity.



Reorder String Equality Check

To avoid NullPointerExceptions, String literals should be placed on the left side when checking for equality.

Issues Fixed: 2

Time Saved: 20 Minutes

StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java)

2 Differences

Original code

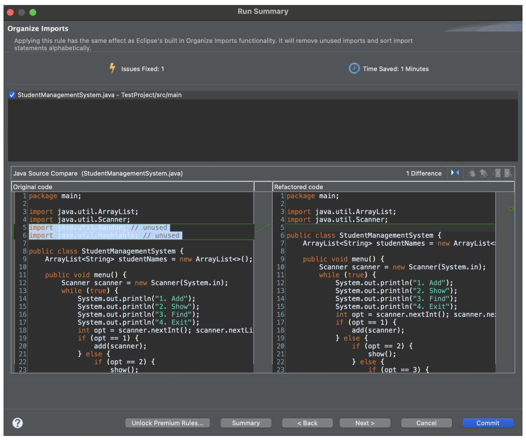
```
d show() {
 for (int i = 0; i < studentNames.size(); i++) {
     if (!(studentNames.get(i) == null)) {
         if (studentNames.get(i).equals("")) {
             System.out.println("Empty name");
         } else {
             System.out.println("Name: " + studentNames
         }
     }
     System.out.println("ID: " + studentIds.get(i));
     System.out.println("GPA: " + studentGPAs.get(i));
 }


d find(Scanner scanner) {
 System.out.println("Search name:");
 String q = scanner.nextLine();
 if (q == null || q.equals("")) {
     throw new IllegalArgumentException("Bad input");
 }
```

Refactored code

```
d show() {
 for (int i = 0; i < studentNames.size(); i++) {
     if (!(studentNames.get(i) == null)) {
         if ("".equals(studentNames.get(i))) {
             System.out.println("Empty name");
         } else {
             System.out.println("Name: " + studentN
         }
     }
     System.out.println("ID: " + studentIds.get(i))
     System.out.println("GPA: " + studentGPAs.get(i
 }


d find(Scanner scanner) {
 System.out.println("Search name:");
 String q = scanner.nextLine();
 if (q == null || "".equals(q)) {
     throw new IllegalArgumentException("Bad input"
 }
```

Figure 5: Reorder String Equality Checks

Rule: Reorder String Equality Checks. jSparrow rearranged `variable.equals("")` into `"".equals(variable)` to prevent potential `NullPointerExceptions` when the variable is null. This is a common best practice in Java, particularly when working with user input or external data. The tool correctly applied this fix to multiple locations in the code.



Organize Imports

Applying this rule has the same effect as Eclipse's built in Organize Imports functionality. It will remove unused imports and sort import statements alphabetically.

Issues Fixed: 1

Time Saved: 1 Minutes

StudentManagementSystem.java - TestProject/src/main

Java Source Compare (StudentManagementSystem.java)   1 Difference

Original code

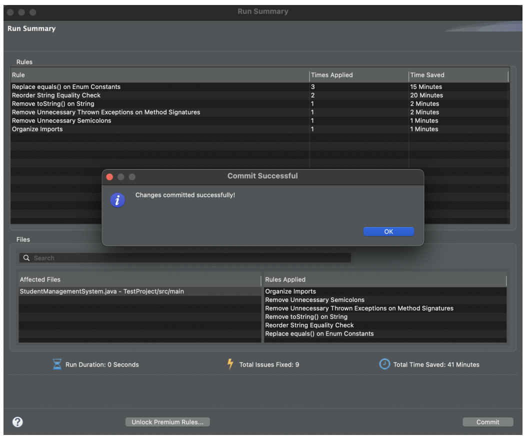
```java
package main;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.Random; // unused
import java.util.Hashtable; // unused

public class StudentManagementSystem {
    ArrayList<String> studentNames = new ArrayList<>();

    public void menu() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("1. Add");
            System.out.println("2. Show");
            System.out.println("3. Find");
            System.out.println("4. Exit");
            int opt = scanner.nextInt(); scanner.nextLi
            if (opt == 1) {
                add(scanner);
            } else {
                if (opt == 2) {
                    show();
```

Refactored code

```java
package main;

import java.util.ArrayList;
import java.util.Scanner;

public class StudentManagementSystem {
    ArrayList<String> studentNames = new ArrayList<

    public void menu() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("1. Add");
            System.out.println("2. Show");
            System.out.println("3. Find");
            System.out.println("4. Exit");
            int opt = scanner.nextInt(); scanner.ne
            if (opt == 1) {
                add(scanner);
            } else {
                if (opt == 2) {
                    show();
                } else {
                    if (opt == 3) {
```

Figure 6: Organize Imports – Remove Unused Imports

Rule: Organize Imports. Unused imports for `Random` and `Hashtable` were removed. This reduces code size and avoids misleading developers into thinking those classes are used. The remaining imports were sorted alphabetically, improving readability and adherence to conventions.



Run Summary

Rules

Rule	Times Applied	Time Saved
Replace equals() on Enum Constants	3	15 Minutes
Reorder String Equality Check	2	20 Minutes
Remove toString() on String	1	2 Minutes
Remove Unnecessary Thrown Exceptions on Method Signatures	1	2 Minutes
Remove Unnecessary Semicolons	1	1 Minutes
Organize Imports	1	1 Minutes

Commit Successful

Changes committed successfully!

Files

Affected Files	Rules Applied
StudentManagementSystem.java - TestProject/src/main	Organize Imports
	Remove Unnecessary Semicolons
	Remove Unnecessary Thrown Exceptions on Method Signatures
	Remove toString() on String
	Reorder String Equality Check
	Replace equals() on Enum Constants

Run Duration: 0 Seconds   Total Issues Fixed: 9   Total Time Saved: 41 Minutes

Figure 7: Commit Summary by jSparrow

Summary of Refactorings. As seen in Figure 7, a total of 9 issues were fixed with an estimated time saving of 41 minutes. All fixes were applied to a single file. The commit window confirms that each rule was applied consistently, and the tool grouped the changes logically to allow review before committing.

5 Conclusion

As shown in Figure 7, jSparrow reported 9 issues fixed, saving an estimated 41 minutes. While it successfully handled several critical refactorings like enum comparison, import cleanup, and redundant operations, it missed some nuanced cases like multiple statements on a single line with excessive semicolons. Nonetheless, for a free tool, it provides valuable assistance in applying consistent code transformations and improving maintainability. The interface is user-friendly, allowing preview and confirmation before applying changes.