# Software Engineering Testing Lab

**Name:** Harshit Kapadia

**Id:** 202201486

# Code Inspection:

Below is a breakdown of issues discovered during the code inspection, categorized by type.

## First 200 Lines Analysis

### Category A: Data Reference Issues

- **Uninitialized Variables**:
  - Variables such as name, gender, age, and phone_no are declared but might be used before being properly assigned values, which could lead to runtime errors.
- **Array Boundaries**:
  - Arrays like char specialization[100] and char name[100] do not implement explicit bounds checking, making them vulnerable to buffer overflow attacks.

### Category B: Data Declaration Issues

- **Implicit Declarations**:
  - Variables like adhaar and identification_id should be clearly declared and initialized with appropriate data types to prevent errors during usage.
- **Array Initialization**:

- String arrays such as `char specialization[100]` and `char gender[100]` could benefit from default initialization to avoid working with undefined or garbage values.

## Category C: Computation Errors

- **Inconsistent Data Handling in Computations**:
  - Since variables like `phone_no` and `adhaar` represent numeric strings, ensure that they are consistently treated as strings to avoid unexpected behavior when performing operations or validations.

## Category E: Control-Flow Issues

- **Risky Use of `goto` Statements**:
  - Using `goto` (e.g., `goto C;`) in validation routines for Aadhaar and mobile numbers can lead to endless loops or unpredictable control flow. It is recommended to use `while` loops with clear termination conditions for improved code structure.

## Category F: Interface Issues

- **Parameter Mismatch**:
  - Functions such as `add_doctor()` and `display_doctor_data()` need to ensure that the number and types of parameters passed align correctly with their definitions to prevent interface errors.

## Category G: Input/Output Issues

- **File Handling Issues**:
  - Operations on files (e.g., `Doctor_Data.data`) should ensure files are successfully opened and always closed after use to avoid issues like memory leaks. Additionally, exceptions for failed file operations (e.g., file not found) should be handled explicitly to prevent program crashes.

---

# Second 200 Lines Analysis

## Category A: Data Reference Issues

- **File Handling Deficiencies**:
  - Files such as Doctor_Data.dat and Patient_Data.dat are frequently used without proper error handling for potential issues (e.g., file not found). It is essential to handle such exceptions to maintain system reliability.

## Category B: Data Declaration Issues

- **Array Size Limitations**:
  - Arrays like name[100], specialization[100], and gender[10] may cause buffer overflows if user inputs exceed the predefined length. Consider implementing input length checks to prevent overflow errors.

## Category C: Computation Errors

- **Vaccine Stock Calculation Vulnerabilities**:
  - The display_vaccine_stock() function calculates the total vaccine stock across multiple centers but lacks safeguards against negative inputs or integer overflows. Adding these checks would prevent potential miscalculations.

## Category E: Control-Flow Issues

- **Overuse of goto Statements**:
  - In functions like add_doctor() and add_patient_data(), multiple goto statements are used to revalidate inputs. These could be replaced with structured loop constructs like while or do-while to improve readability and reduce complexity.

## Category F: Interface Issues

- **Faulty String Comparisons**:
    - In search_doctor_data(), string comparisons involving identification_id and sidentification_idrely on .compare(). Ensure all string comparisons are carefully managed to avoid logical errors and inconsistencies.

## Category G: Input/Output Issues

- **Missing File Closure**:
    - Files opened in search_center() and display_vaccine_stock() should always be closed after use to avoid resource leaks or file locking issues.

## Control Flow Suggestion

Replacing goto statements with well-structured loops in the input validation sections will improve maintainability and reduce the chances of hard-to-diagnose bugs. Loops with clear exit conditions ensure smoother flow control and more readable code.

## Third 200 Lines Inspection

## Category A: Data Reference Issues

- **File Handling Validation**:
    - In functions like add_vaccine_stock() and display_vaccine_stock(), ensure that files for vaccine centers (e.g., center1.txt, center2.txt) are successfully opened before proceeding with operations. Incorporate error checking to catch any file access failures.

## Category B: Data Declaration Issues

- **Inconsistent Data Types**:
    - The variables adhaar and phone_no are expected to hold numeric string values but are treated inconsistently across multiple functions.

Ensure uniform handling of these values to avoid misinterpretation as integers.

## Category C: Computation Errors

- **Vaccine Stock Calculation**:
  - In display_vaccine_stock(), incorrect stock totals may occur if values are uninitialized or negative. Add safeguards to initialize all relevant variables and validate the stock numbers before summation.

## Category E: Control-Flow Issues

- **Overuse of goto Statements**:
  - The use of goto in functions like search_doctor_data() and add_doctor() complicates the control flow. Adopting structured loops such as while or for would make the code easier to understand and maintain.

## Category F: Interface Issues

- **Parameter Consistency**:
  - Verify that functions like search_by_aadhar() maintain consistent parameter expectations across all modules to prevent mismatches and runtime errors.

## Category G: Input/Output Issues

- **Improper File Closing**:
  - Files like Doctor_Data.data are sometimes left open in certain code paths. Make sure every file operation ends with proper closing to avoid resource exhaustion or file lock issues.

---

## Fourth 200 Lines Inspection

## Category A: Data Reference Issues

- **Uninitialized Variables**:
  - Functions such as update_patient_data(), show_patient_data(), and applied_vaccine() should initialize variables like maadhaar and file streams explicitly to prevent unexpected behavior due to uninitialized data.

## Category B: Data Declaration Issues

- **Array Length Limits**:
  - Character arrays like gender[10] and adhaar[12] risk overflow if input values exceed the defined size. Input validation is necessary to mitigate this.

## Category C: Computation Errors

- **Incorrect Dose Incrementation**:
  - In update_patient_data(), using dose++ directly could lead to invalid counts without checks. Validate the dose range before incrementing to ensure data accuracy.

## Category E: Control-Flow Issues

- **Excessive Reliance on goto**:
  - The use of goto in search_doctor_data() and add_patient_data() leads to tangled logic. Replacing these with loops would streamline the code structure and enhance readability.

## Category F: Interface Issues

- **String Comparison Errors**:
  - Functions like search_by_aadhar() compare string variables (e.g., adhaar.compare(sadhaar)) but may not account for edge cases. Implement consistent string handling to prevent errors.

## Category G: Input/Output Issues

- **File Handling Vulnerabilities**:
    - Files like Patient_Data.dat and Doctor_Data.dat are opened in several functions without adequate error handling. Ensure proper exception handling to avoid runtime crashes.

---

## Fifth 200 Lines Inspection

## Category A: Data Reference Issues

- **Uninitialized Variables**:
    - Variables such as maadhaar in update_patient_data() and other fields in search_doctor_data() should be initialized to prevent usage before assignment.

## Category B: Data Declaration Issues

- **Array Boundary Issues**:
    - Arrays such as gender[10] risk buffer overflows if input exceeds the allocated space. Validate input lengths to avoid this issue.

## Category C: Computation Errors

- **Unchecked Dose Increment**:
    - In update_patient_data(), the dose++ operation is performed without validation, which can result in invalid dose counts. Add proper checks to avoid such errors.

## Category E: Control-Flow Issues

- **Redundant Use of goto**:

- Functions like search_doctor_data() and add_doctor() contain multiple goto statements, making the logic hard to follow. Consider using loops for better maintainability.

## Category F: Interface Issues

- **Parameter Handling Issues**:
  - Ensure that parameters, such as those in search_by_aadhar(), are passed correctly with consistent types across all modules.

## Category G: Input/Output Issues

- **Unclosed File Operations**:
  - Some file operations in Patient_Data.dat and Doctor_Data.data leave files open in specific code paths. Always close files after use to avoid resource leaks.

---

## Final 300 Lines Inspection

## Category A: Data Reference Issues

- **File Access without Error Handling**:
  - Files like center1.txt, center2.txt, and center3.txt are accessed in add_vaccine_stock() and display_vaccine_stock() without adequate error handling. Add checks to handle file access errors gracefully.

## Category B: Data Declaration Issues

- **Uninitialized Variables**:
  - Variables like sum_vaccine_c1, sum_vaccine_c2, and sum_vaccine_c3 should be initialized to prevent undefined behavior during vaccine stock calculations.

## Category C: Computation Errors

- **Invalid Stock Values**:
    - In add_vaccine_stock(), ensure that stock values are always positive and valid to prevent incorrect calculations when displayed.

### Category E: Control-Flow Issues

- **Overuse of goto Statements**:
    - In functions like add_doctor() and add_patient_data(), the frequent use of goto makes the code challenging to maintain. Loops would offer a cleaner and more structured approach.

### Category G: Input/Output Issues

- **Inconsistent File Closure**:
    - In some file-handling branches, files are not closed correctly, risking memory leaks. Ensure every opened file is closed after operations and handle exceptions where necessary.

# DEBUGGING

### 1. Armstrong Number Program

- **Error:** Incorrect computation of the remainder.
- **Fix:** Use breakpoints to validate remainder calculation logic.

*Corrected Code:*

```
class Armstrong {
public static void main(String args[]) {
  int num = Integer.parseInt(args[0]); int n = num, check = 0, remainder; while (num > 0) {
  remainder = num % 10;
  check += Math.pow(remainder, 3); num /= 10;
}
if (check == n) {
  System.out.println(n + " is an Armstrong Number"); } else {
    System.out.println(n + " is not an Armstrong Number"); }
} }
```

## 2. GCD and LCM Program

- **Errors:**
    1. Incorrect while loop condition in GCD.
    2. Incorrect LCM logic.
- **Fix:** Add breakpoints to validate GCD and LCM operations.

*Corrected Code:*

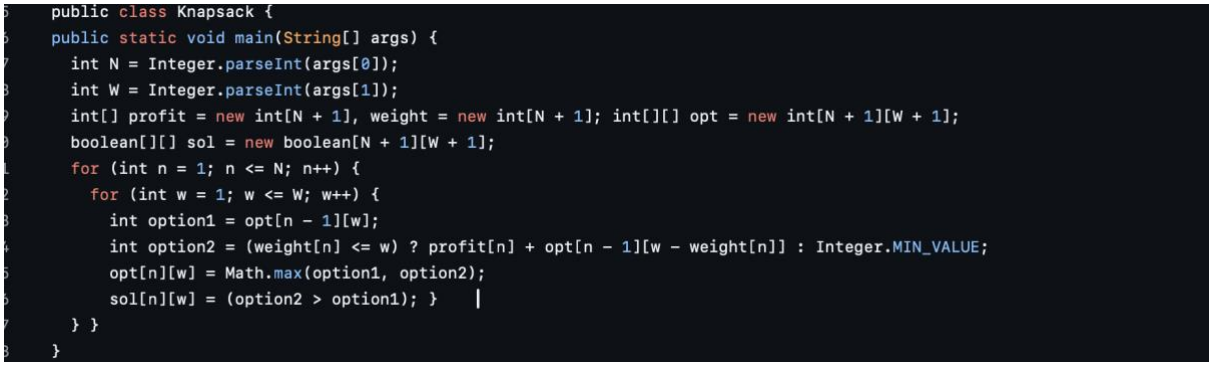
```
class Armstrong {
public static void main(String args[]) {
  int num = Integer.parseInt(args[0]); int n = num, check = 0, remainder; while (num > 0) {
  remainder = num % 10;
  check += Math.pow(remainder, 3); num /= 10;
}
if (check == n) {
  System.out.println(n + " is an Armstrong Number"); } else {
    System.out.println(n + " is not an Armstrong Number"); }
} }
import java.util.Scanner; public class GCD_LCM {
static int gcd(int x, int y) { while (y != 0) {
  int temp = y; y = x % y;
  x = temp;
}
return x; }
static int lcm(int x, int y) { return (x * y) / gcd(x, y);
}
public static void main(String args[]) {
  Scanner input = new Scanner(System.in); System.out.println("Enter the two numbers: "); int x = input.nextInt();
  int y = input.nextInt();
  System.out.println("The GCD of two numbers is: " + gcd(x, y)); System.out.println("The LCM of two numbers is: " + lcm(x, y)); input.close();
} }

```

## 3. Knapsack Program

- **Error:** Incorrect `n` increment in the loop.
- **Fix:** Add breakpoints to check loop behavior.

*Corrected Code:*



```
public class Knapsack {
public static void main(String[] args) {
  int N = Integer.parseInt(args[0]);
  int W = Integer.parseInt(args[1]);
  int[] profit = new int[N + 1], weight = new int[N + 1]; int[][] opt = new int[N + 1][W + 1];
  boolean[][] sol = new boolean[N + 1][W + 1];
  for (int n = 1; n <= N; n++) {
    for (int w = 1; w <= W; w++) {
      int option1 = opt[n - 1][w];
      int option2 = (weight[n] <= w) ? profit[n] + opt[n - 1][w - weight[n]] : Integer.MIN_VALUE;
      opt[n][w] = Math.max(option1, option2);
      sol[n][w] = (option2 > option1); }
  } }
}
```

## 4. Magic Number Program

- **Errors:**
    1. Incorrect condition in inner `while` loop.

2. Missing semicolons.

- **Fix:** Validate while loop logic with breakpoints.

```java
import java.util.Scanner;

public class MagicNumberCheck {
    public static void main(String[] args) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to check:");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while (num > 9) {
            sum = num;
            int s = 0;
            while (sum > 0) {
                s += sum % 10;
                sum /= 10;
            }
            num = s;
        }
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

## 5. Merge Sort Program

- **Errors:**
    1. Incorrect array splitting logic.
    2. Incorrect merge operation inputs.
- **Fix:** Add breakpoints to debug merging and splitting.

*Corrected Code:*

```java
import java.util.Arrays;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("Before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("After: " + Arrays.toString(list));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        System.arraycopy(array, 0, left, 0, size1);
        return left;
    }

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        System.arraycopy(array, size1, right, 0, size2);
        return right;
    }

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0, i2 = 0;
        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {
                result[i] = left[i1++];
            } else {
                result[i] = right[i2++];
            }
        }
    }
}
```

## 6. Multiply Matrices Program

- **Errors:**
    1. Incorrect loop indices.
    2. Wrong error message.

- **Fix:** Debug matrix multiplication logic with breakpoints.

*Corrected Code:*

```
class MatrixMultiplication {
    public static void main(String args[]) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter rows and columns of the first matrix:");
        int m = in.nextInt(), n = in.nextInt();
        int[][] first = new int[m][n];
        System.out.println("Enter elements of the first matrix:");
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                first[i][j] = in.nextInt();

        System.out.println("Enter rows and columns of the second matrix:");
        int p = in.nextInt(), q = in.nextInt();
        if (n != p) {
            System.out.println("Matrices cannot be multiplied.");
            return;
        }

        int[][] second = new int[p][q], product = new int[m][q];
        System.out.println("Enter elements of the second matrix:");
        for (int i = 0; i < p; i++)
            for (int j = 0; j < q; j++)
                second[i][j] = in.nextInt();

        for (int i = 0; i < m; i++)
            for (int j = 0; j < q; j++) {
                for (int k = 0; k < n; k++)
                    product[i][j] += first[i][k] * second[k][j];
            }

        System.out.println("Product of matrices:");
        for (int[] row : product) {
            for (int val : row) System.out.print(val + " ");
            System.out.println();
        }
    }
}
```

# 7. Quadratic Probing Hash Table Program

## Errors:

1. **Typos** in the `insert`, `remove`, and `get` methods.
2. **Incorrect rehashing logic.**

## Corrected Code:

```java
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys, vals;

    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public void insert(String key, String val) {
        int tmp = hash(key), i = tmp, h = 1;
        do {
            if (keys[i] == null) {
                keys[i] = key;
                vals[i] = val;
                currentSize++;
                return;
            }
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
            i = (i + h * h++) % maxSize;  // Corrected rehash logic
        } while (i != tmp);
    }

    public String get(String key) {
        int i = hash(key), h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key)) return vals[i];
            i = (i + h * h++) % maxSize;
        }
        return null;
    }

    public void remove(String key) {
        if (!contains(key)) return;
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxSize;
        keys[i] = vals[i] = null;  // Set both key and value to null
    }

    private boolean contains(String key) {
        return get(key) != null;
    }

    private int hash(String key) {
        return key.hashCode() % maxSize;
    }
}

public class HashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the capacity of the hash table: ");
        int capacity = scan.nextInt();
        QuadraticProbingHashTable hashTable = new QuadraticProbingHashTable(capacity);

        hashTable.insert("key1", "value1");
        System.out.println("Value: " + hashTable.get("key1"));
    }
}
```

# 8. Sorting Array Program

### Errors:

1. **Incorrect class name** with an extra space.
2. **Incorrect loop condition** and extra semicolon.

### Corrected Code:

```java
import java.util.Arrays;
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        n = s.nextInt();
        int[] a = new int[n];

        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) a[i] = s.nextInt();

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        System.out.println("Sorted Array: " + Arrays.toString(a));
    }
}
```

# 9. Stack Implementation Program

## Errors:

1. **Incorrect top increment** (`top--` instead of `top++` in `push`).
2. **Incorrect loop condition** in `display`.
3. **Missing pop method.**

### Corrected Code:

```java
public class StackMethods {
    private int top;
    private int[] stack;

    public StackMethods(int size) {
        stack = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == stack.length - 1) {
            System.out.println("Stack full");
        } else {
            stack[++top] = value;  // Increment top before assigning value
        }
    }

    public void pop() {
        if (top == -1) {
            System.out.println("Stack empty");
        } else {
            top--;
        }
    }

    public void display() {
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        StackMethods stack = new StackMethods(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.display();  // Output: 10 20 30
        stack.pop();
        stack.display();  // Output: 10 20
    }
}
```

# 10. Tower of Hanoi Program

### Error:

1. **Incorrect increment/decrement** in recursive calls.

```java
public class TowerOfHanoi {
    public static void main(String[] args) {
        int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');
    }

    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) {
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            doTowers(topN - 1, from, to, inter);  // Corrected recursive call order
            System.out.println("Disk " + topN + " from " + from + " to " + to);
            doTowers(topN - 1, inter, from, to);
        }
    }
}
```

# STATIC ANALYSIS TOOL:

**Program inspected:** 1300 lines of code
**Analysis Tool:** Cppcheck

## Results and Warnings:

**Information:**
Cppcheck reported the following standard headers were **not found**, but these are **not required** for accurate static analysis:

<stdio.h>, <stdlib.h>,<sys/types.h>,<sys/stat.h>, <unistd.h>, <dirent.h>, <fcntl.h>, <libgen.h>, <errno.h>, <string.h>, <windows.h>, <iostream>, <fstream>, <conio.h>, <iomanip>, <cstdlib>, <string>

**[202201486_Lab3_2.c]:**

- [Line 0]: (information) Limiting analysis of branches. Use --check-level=exhaustive for complete branch coverage.
- [Line 116, 120, 126, 127, 133]: (warning) **scanf()** used without field width limits; this can crash with large inputs.
- [Line 34]: (style) The scope of variable **'ch'** can be reduced.
- [Line 115]: (style) The scope of variable **'path2'** can be reduced.
- [Line 16]: (style) Parameter **'file'** can be declared as a pointer to const.
- [Line 55]: (style) Variable **'direntp'** can be declared as a pointer to const.
- [Line 40]: (warning) **fgetc()** return value stored in char variable and compared with EOF, which can cause issues.

**[202201486_Lab3_3.c]:**

- [Lines 1-5]: (information) Missing standard headers detected: <stdio.h>, <stdlib.h>, <sys/types.h>, <sys/stat.h>, <unistd.h>.

**[202201486_Lab3_1.c]:**

- [Lines 1-9]: (information) Same missing headers as above.
- [Line 29]: (style) The scope of variable **'ch'** can be reduced.
- [Line 11]: (style) Parameter **'file'** can be declared as a pointer to const.
- [Line 50]: (style) Variable **'direntp'** can be declared as a pointer to const.
- [Line 35]: (warning) **fgetc()** return value stored in char and compared with EOF, which can cause errors.

**[Covid-Management-System.cpp]:**

- [Lines 4-12]: (information) Missing headers include <iostream>, <cstring>, <windows.h>, <fstream>, <conio.h>, <iomanip>, <cstdlib>, <string>, <unistd.h>.
- [Line 562, 565, 614, 1121]: (portability) Calling **fflush()** on stdin may cause undefined behavior on non-Linux systems.
- [Lines 538-1317]: (style) Multiple instances of **C-style pointer casting** detected.
- [Line 427, 443, 459, 892]: (style) Unnecessary consecutive **return, break, continue, goto**, or **throw**statements.

- [Line 306]: (style) The scope of variable **'usern'** can be reduced.
- [Line 48 -> 277]: (style) Local variable **'user'** shadows an outer function.
- [Line 40 -> 304]: (style) Local variable **'c'** shadows an outer variable.
- [Line 275]: (performance) Function parameter **'str'** should be passed as a **const reference**.
- [Line 277, 304]: (style) **Unused variables**: 'user', 'c'.