IT314
Software Engineering

Jay Malaviya
202101048

Lab 9
Program Inspection and Debugging

Code 1: Armstrong Number

Category A: An Error Checklist for Inspections

● No errors in this category.

Which category of program inspection would you find more effective?

● Category A: An Error Checklist for Inspections.

Which type of error you are not able to identify using the program inspection?

● No errors are found in this code.

Is the program inspection technique worth applicable?

● Yes, program inspection can be worth applying to this code to ensure correctness.

Debugging

1. Errors:

- Error 1: Incorrect usage of post-increment and post-decrement operators (`topN++`, `inter--`).

- Error 2: Incorrect order of parameters in the recursive call to `doTowers`.

2. Breakpoints Needed: At least two breakpoints are needed to address these errors.

3. Steps Taken to Fix Errors:

- Replace `topN++` with `topN--`.

- Swap the order of `from` and `inter` in the recursive call.

```
public class Armstrong {
   public static void main(String[] args) {
      if (args.length == 0) {
```

```
            System.out.println("Please provide an integer
input.");
            return;
        }

        try {
            int num = Integer.parseInt(args[0]);
            int originalNumber = num; // used to check at the
last time
            int check = 0;
            int remainder;

            while (num > 0) {
                remainder = num % 10;
                check += Math.pow(remainder, 3);
                num /= 10;
            }

            if (check == originalNumber) {
                System.out.println(originalNumber + " is an
Armstrong Number");
            } else {
                System.out.println(originalNumber + " is not an
Armstrong Number");
            }
        } catch (NumberFormatException e) {
            System.out.println("Please provide a valid integer
input.");
```

```
        }
    }
}
```

Code 2 (Tower of Hanoi):

Category E: Control-Flow Errors

● There's a logical error in the code. The recursive call in the doTowers method is not properly structured.

Which category of program inspection would you find more effective?

● Category E: Control-Flow Errors.

Which type of error you are not able to identify using the program inspection?

● Other types of errors (e.g., data declaration or computation errors) are not present in this code.

Is the program inspection technique worth applicable?

● Yes, program inspection is worth applying to this code to correct the

control-flow error.

Debugging:

1. Errors:

- Error 1: In the `push` method, change `top--` to `top++` to increment the top before pushing the value.

- Error 2: In the `display` method, change the loop condition from `for(int i=0; i > top; i++)` to `for (int i = 0; i < top; i++)` to correctly display the stack.

2. Breakpoints Needed: Two breakpoints are needed to address these errors.

3. Steps Taken to Fix Errors:

- In the `push` method, change `top--` to `top++`.

- In the `display` method, change the loop condition to use `<` instead of `>`.

Corrected code

```
public class MainClass {
   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("Move disk 1 from " + from + " to "
+ to);
      } else {
         doTowers(topN - 1, from, to, inter);
         System.out.println("Move disk " + topN + " from " +
from + " to " + to);
         doTowers(topN - 1, inter, from, to);
      }
   }
}
```

Code 3 Knapsack:

Category B: Data-Declaration Errors

● There is a mistake in the loop indices. It should be for (int n = 1; n <= N; n++) instead of for (int n = 1; n <= N; n++).

Which category of program inspection would you find more effective?

● Category B: Data-Declaration Errors.

Which type of error you are not able to identify using the program inspection?

● There are no data-declaration errors in this code.

Is the program inspection technique worth applicable?

● Yes, program inspection can be applied to correct the data-declaration error.

Debugging:1. Errors:

- Error 1: In the first for loop, change `for (int i = 0; i >= n; i++);` to `for (int i = 0; i < n; i++)` to

correctly iterate through the array.

- Error 2: In the second for loop, change `for (int i = 0; i >= n; i++)` to `for (int i = 0; i < n; i++)`

to correctly iterate through the array.

2. Breakpoints Needed: Two breakpoints are needed to address these errors.

3. Steps Taken to Fix Errors:

- In the first for loop, change the condition to `i < n` to iterate through the array.

- In the second for loop, change the condition to `i < n` to iterate through the array.

Corrected code

```
import java.util.Random;


public class Knapsack {
   public static void main(String[] args) {
      if (args.length < 2) {
         System.out.println("Please provide two integer arguments for the number of items and maximum weight of the knapsack.");
         return;
      }


      int N = Integer.parseInt(args[0]); // number of items
      int W = Integer.parseInt(args[1]); // maximum weight of knapsack
      int[] profit = new int[N + 1];
      int[] weight = new int[N + 1];
      Random random = new Random();
      // Generate random instance, items 1..N
      for (int n = 1; n <= N; n++) {
         profit[n] = random.nextInt(1000);
         weight[n] = random.nextInt(W);
      }
      // opt[n][w] = max profit of packing items 1..n with weight limit w
      // sol[n][w] = does opt solution to pack items 1..n with weight limit w include item n?
      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++) {
        // Don't take item n
        int option1 = opt[n - 1][w];
        // Take item n
        int option2 = Integer.MIN_VALUE;
        if (weight[n] <= w) {
            option2 = profit[n] + opt[n - 1][w - weight[n]];
        }
        // Select the better of two options
        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}
// Determine which items to take
boolean[] take = new boolean[N + 1];
for (int n = N, w = W; n > 0; n--) {
    if (sol[n][w]) {
        take[n] = true;
        w -= weight[n];
    } else {
        take[n] = false;
    }
}
// Print results
System.out.println("Item\tProfit\tWeight\tTake");
```

```
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n]
+ "\t" + take[n]);
        }
    }
}
```

Code 4 (Stack implementation in java):

Category B: Data-Declaration Errors

● There are issues with the loop indices. It should be for(int i = 0; i > top; i++) instead of for(int i = 0; i > top; i++).

Category G: Input / Output Errors

● In the pop method, there's a missing newline character in the output.

Which category of program inspection would you find more effective?

● Category B: Data-Declaration Errors.Which type of error you are not able to identify using the program inspection?

● Data declaration and input/output errors are identified, but there might be other logical errors that aren't captured by this inspection.

Is the program inspection technique worth applicable?

● Yes, program inspection can be applied to correct the data-declaration and

input/output errors, but further testing and inspection may be needed to identify other issues.

In summary, program inspection is a valuable technique to identify and correct various types of errors in code. It's essential to apply it to different categories of errors to ensure code quality and correctness. However, it may not catch all logical errors, so thorough

testing is still required.

Debugging:

1. Errors:

- Error 1: In the first for loop, change `for (int i = 0; i >= n; i++);` to `for (int i = 0; i < n; i++)` to correctly iterate through the array.

- Error 2: In the second for loop, change `for (int i = 0; i >= n; i++)` to `for (int i = 0; i < n; i++)` to correctly iterate through the array.

2. Breakpoints Needed: Two breakpoints are needed to address these errors.

3. Steps Taken to Fix Errors:

- In the first for loop, change the condition to `i < n` to iterate through the array.

- In the second for loop, change the condition to `i < n` to iterate through the array.

Corrected code

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

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

    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;
            stack[top] = value;
```

```
        }
    }

    public void pop() {
        if (!isEmpty()) {
            top--;
        } else {
            System.out.println("Can't pop... stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
```

Code 5: Quadratic Probing Hash Table

Category B: Data-Declaration Errors

● There is a syntax error in the line i + = (i + h / h--) % maxSize;. The correct form should be i += (i + h / h--) % maxSize;.

Category E: Control-Flow Errors

● There's an issue with the control flow of the insert and get methods. These methods use quadratic probing to handle collisions, and the probing logic appears to be incorrect.

Which category of program inspection would you find more effective?

● Category E: Control-Flow Errors.

Which type of error you are not able to identify using the program inspection?

● Data-declaration and control-flow errors are identified, but there might be other logical errors in the hashing and probing logic that aren't captured by this inspection.

Is the program inspection technique worth applicable?

● Yes, program inspection can be applied to correct data-declaration and control-flow errors, but the hashing and probing logic need further review to ensure correctness.

Debugging:

1. Errors:- Error 1: In the `insert` method, change `i + = (i + h / h--) % maxSize;` to `i = (i + h *

h++) % maxSize;`.

- Error 2: In the `get` method, change the loop condition to `for (int j = 0; j < maxSize;

j++)` to ensure it doesn't run indefinitely.

- Error 3: In the `remove` method, change the rehash loop to `for (int j = 0; j < maxSize;

j++)` to ensure it rehashes properly.

2. Breakpoints Needed: At least three breakpoints are needed to address these errors.

Corrected code

```
import java.util.Scanner;


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

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

   public void makeEmpty() {
      currentSize = 0;
      keys = new String[maxSize];
      vals = new String[maxSize];
   }

   public int getSize() {
```

```
    return currentSize;
}

public boolean isFull() {
    return currentSize == maxSize;
}

public boolean isEmpty() {
    return getSize() == 0;
}

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

private int hash(String key) {
    return (key.hashCode() & 0x7FFFFFFF) % maxSize;
}

public void insert(String key, String val) {
    if (isFull()) {
        System.out.println("Hash table is full, cannot insert.");
        return;
    }
    int i = hash(key);
    int h = 1;
```

```
    while (keys[i] != null) {
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i = (i + h * h) % maxSize;
        h++;
    }
    keys[i] = key;
    vals[i] = val;
    currentSize++;
}

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

```
    public void remove(String key) {
        if (!contains(key)) {
            return;
        }
        int i = hash(key);
        int h = 1;
        while (!key.equals(keys[i])) {
            i = (i + h * h) % maxSize;
            h++;
        }
        keys[i] = null;
        vals[i] = null;
        currentSize--;
        for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h) %
maxSize) {
            String tmp1 = keys[i];
            String tmp2 = vals[i];
            keys[i] = null;
            vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
    }

    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++) {
```

```
            if (keys[i] != null) {
                System.out.println(keys[i] + " " + vals[i]);
            }
        }
        System.out.println();
    }
}

public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");
        QuadraticProbingHashTable qpht = new
QuadraticProbingHashTable(scan.nextInt());
        char ch;
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert");
            System.out.println("2. remove");
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");
            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value");
```

```
                qpht.insert(scan.next(), scan.next());
                break;
            case 2:
                System.out.println("Enter key");
                qpht.remove(scan.next());
                break;
            case 3:
                System.out.println("Enter key");
                System.out.println("Value = " +
qpht.get(scan.next()));
                break;
            case 4:
                qpht.makeEmpty();
                System.out.println("Hash Table Cleared\n");
                break;
            case 5:
                System.out.println("Size = " + qpht.getSize());
                break;
            default:
                System.out.println("Wrong Entry \n");
                break;
          }
          qpht.printHashTable();
          System.out.println("\nDo you want to continue (Type
y or n) \n");
          ch = scan.next().charAt(0);
       } while (ch == 'Y' || ch == 'y');
```

```
    }
}
```

Code 6: Matrix Multiplication

Category B: Data-Declaration Errors

● No errors in this category.

Category C: Computation Errors

● There is a computation error in the matrix multiplication loop. The index calculations for accessing elements are incorrect.

Which category of program inspection would you find more effective?

● Category C: Computation Errors.

Which type of error you are not able to identify using the program inspection?

● There are no data-declaration errors, but the computation errors in the matrix multiplication logic need to be fixed.

Is the program inspection technique worth applicable?

● Yes, program inspection can be applied to correct computation errors in the matrix multiplication logic.

Debugging:

1. Errors:

- Error 1: In the nested loop that calculates the product of matrices, change `sum = sum +

first[c-1][c-k]*second[k-1][k-d];` to `sum = sum + first[c][k] * second[k][d];`.

- Error 2: Initialize variables `c`, `d`, and `k` properly within the for loops.

2. Breakpoints Needed: At least two breakpoints are needed to address these errors.

Corrected Code

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

```
class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q;
      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number of rows and
columns of the first matrix");
      m = in.nextInt();
      n = in.nextInt();
      int first[][] = new int[m][n];
      System.out.println("Enter the elements of the first
matrix");
      for (int c = 0; c < m; c++) {
         for (int d = 0; d < n; d++) {
            first[c][d] = in.nextInt();
         }
      }
      System.out.println("Enter the number of rows and
columns of the second matrix");
      p = in.nextInt();
      q = in.nextInt();
      if (n != p) {
         System.out.println("Matrices with entered orders
can't be multiplied with each other.");
      } else {
         int second[][] = new int[p][q];
         int multiply[][] = new int[m][q];
```

```
        System.out.println("Enter the elements of the
second matrix");
        for (int c = 0; c < p; c++) {
           for (int d = 0; d < q; d++) {
              second[c][d] = in.nextInt();
           }
        }
        for (int c = 0; c < m; c++) {
           for (int d = 0; d < q; d++) {
              int sum = 0;
              for (int k = 0; k < p; k++) {
                 sum += first[c][k] * second[k][d];
              }
              multiply[c][d] = sum;
           }
        }
        System.out.println("Product of entered matrices:");
        for (int c = 0; c < m; c++) {
           for (int d = 0; d < q; d++) {
              System.out.print(multiply[c][d] + "\t");
           }
           System.out.println();
        }
     }
  }
}
```

Code 7: Magic Number Check

Category A: An Error Checklist for Inspections

● There are syntax errors in this code, such as missing semicolons and incorrect operators in lines while(sum==0) and sum=num.

Category C: Computation Errors

● There's a computation error in the line s=s*(sum/10);. It should be s=s+(sum/10);, and there's another error in sum=num;, which should be sum=0;

Which category of program inspection would you find more effective?

● Category A: An Error Checklist for Inspections.

Which type of error you are not able to identify using the program inspection?

● Data-declaration and computation errors are identified. The inspection also points out the need for correcting syntax errors.

Is the program inspection technique worth applying?

● Yes, program inspection can be applied to correct syntax, data-declaration, and computation errors in this code.

In summary, program inspection is a valuable technique to identify and correct various types of errors in code. It must be applied to different categories of errors to ensure code quality and correctness. However, it may not catch all logical errors, so thorough testing and debugging may be needed.

Code 8 merge short

Category A: An Error Checklist for Inspections

Category E: Control-Flow Errors

Errors:

Control-Flow Error: In the mergeSort method, there are issues with the recursive calls and array splitting. The code int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1); is incorrect. It should be int[] left = leftHalf(array); and int[] right = rightHalf(array);. Also, merge(array, left++, right--); should be merge(array, left, right);.

Control-Flow Error: There is a logical error in the recursive calls to mergeSort. The recursive calls should be made on the left and right arrays, not the original array.

Control-Flow Error: The merge method is not defined in the provided code. This is an undeclared method.

Control-Flow Error: The left and right half methods do not return correct results because they do not split the array correctly.

Which category of program inspection would you find more effective?

● Category E (Control-Flow Errors) would be the most effective category for inspecting and identifying errors in this code.

Which type of error you are not able to identify using the program inspection?

● Some of the specific logical errors, like incorrect recursive calls, would not be

identified using program inspection alone. You must understand the

intended algorithm and logic to identify these issues.

Is the program inspection technique worth applying?

● Yes, program inspection is a valuable technique for identifying errors in code.

However, in more complex cases like this, a deep understanding of the algorithm

and careful manual review of the code is required to identify all errors.

Debugging:

1. Errors:

- Error 1: In the `mergeSort` method, replace `int[] left = leftHalf(array+1);` with `int[] left =

leftHalf(array);` and `int[] right = rightHalf(array-1);` with `int[] right = rightHalf(array);`.

- Error 2: The `merge` method is called in the code but not provided. A correct implementation

of the `merge` method is needed for the code to work.

2. Breakpoints Needed: At least two breakpoints address these errors.

Corrected Code

```
import java.util.*;


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];
    for (int i = 0; i < size1; i++) {
        left[i] = array[i];
    }
    return left;
}

public static int[] rightHalf(int[] array) {
    int size1 = array.length / 2;
    int size2 = array.length - size1;
    int[] right = new int[size2];
    for (int i = 0; i < size2; i++) {
        right[i] = array[i + size1];
```

```
        }
        return right;
    }

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0;
        int 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];
                i1++;
            } else {
                result[i] = right[i2];
                i2++;
            }
        }
    }
}
```

Code 9: Ascending Order

Category: B: Data-Declaration Errors C: Computation Errors

Errors:

Data-Declaration Error: There is a space in the class name. It should be Ascending_Order instead of Ascending_Order.

Computation Error: The loop condition for (int i = 0; i >= n; i++) is incorrect. It should be for (int i = 0; i < n; i++).

Which category of program inspection would you find more effective?

● Category C (Computation Errors) would be the most effective category for inspecting and identifying errors in this code.

Which type of error you are not able to identify using the program inspection?

● Program inspection can identify most errors in this code. However, it might not catch logical errors related to the sorting algorithm if the algorithm's logic is flawed.

Is the program inspection technique worth applying?

● Yes, program inspection is applicable and effective for identifying many errors in this code, especially data declaration and computation errors.

Debugging:

1. Errors:

- Error 1: Change the inner while loop's condition from `while (sum == 0)` to `while (sum > 0)`

to avoid an infinite loop.

- Error 2: Add semicolons at the end of lines with `sum=sum%10` and `s=s*(sum/10)` to fix the

syntax errors.

2. Breakpoints Needed: At least two breakpoints are needed to address these errors.

Corrected Code

```
import java.util.*;


public class MagicNumberCheck {
   public static void main(String args[]) {
      Scanner ob = new Scanner(System.in);
      System.out.println("Enter the number to be checked.");
      int n = ob.nextInt();
      int sum = 0, num = n;
      while (num > 9) {
         sum = num;
         int s = 1; // Updated the initial value of s to 1
         while (sum > 0) {
            s = s * (sum % 10); // Fixed the computation and updated the value of s
            sum = sum / 10; // Fixed the computation
         }
         num = s;
      }
      if (num == 1) {
         System.out.println(n + " is a Magic Number.");
      } else {
         System.out.println(n + " is not a Magic Number.");
      }
```

```
    }
}
```

Code 10: GCD_LCM

Category: D: Comparison Errors E: Control-Flow Errors G: Input / Output ErrorsErrors:

Comparison Error: The condition in the while loop for calculating GCD is incorrect. It should be while(a % b != 0) instead of while(a % b == 0).

Control-Flow Error: There is a logical error in the lcm method. The logic used to calculate LCM is incorrect. It doesn't calculate the LCM correctly.

Which category of program inspection would you find more effective?

● Category D (Comparison Errors) and E (Control-Flow Errors) are the most effective categories for inspecting and identifying errors in this code.

Which type of error you are not able to identify using the program inspection?

● The logical error in the lcm method related to LCM calculation may not be easy to identify using program inspection alone. Understanding the correct LCM algorithm is necessary.

Is the program inspection technique worth applicable?

● Program inspection is applicable for identifying some errors in this code, but it may not be sufficient to catch the logical error in the LCM calculation.

Debugging

1. Errors:

- Error 1: In the `gcd` method, change the while loop condition to `while(a % b != 0)` to find the greatest common divisor correctly.

2. Breakpoints Needed: At least one breakpoint is needed to address these errors.

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number
        while (b != 0) { // Corrected the loop condition
            r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    static int lcm(int x, int y) {
        int a;
        a = (x > y) ? x : y; // a is the greater number
        while (true) {
            if (a % x == 0 && a % y == 0) {
                return a;
            }
            ++a;
        }
    }
```

```
    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();
    }
}
```

In all cases, program inspection can help identify a range of errors, but it's not a substitute for a deep understanding of the algorithms and logic used in the code.