



# CM2307: Object Orientation, Algorithms and Data Structures

REFLECTIONS ON PROGRAM DESIGN AND IMPLEMENTATION USING OBJECT  
ORIENTED PRINCIPLES INVOLVING APPROPRIATE DATA STRUCTURES AND  
ALGORITHMS **(100%)**

Lakshmi Ksheeraja Sikha | 23112887 | 8<sup>th</sup> January 2025

## Introduction

This project's purpose was to design and implement a Network Management System (NMS) that manages network devices and routes between them. The NMS is a software system used to monitor, configure, and manage the network in real-time. It consists of key components such as Network Device Management, Route Management, Monitoring and Diagnostics, Logging and Events Management, and User Interface.

In this assignment, we were expected to design and implement the components of the system using appropriate Object-Oriented principles and data structures. Additionally, we were required to create UML diagrams, including use case diagrams, class diagrams, and sequence diagrams, to visualize the system's architecture and interactions. The implementation was based on the starter code provided, and reflections on observations during the assignment were also included.

## Analysis and Evaluation of design principles and patterns in Network Management Systems (NMS):

The NMS consists of several classes, each responsible for various aspects of the system.

- **NMS.java:** The main class that serves as the entry point for the application, handling user input and orchestrating the functionality of other classes.

```
2  * This is the primary class of the system.
3  * It will be used to launch the system and perform
4  * the necessary actions, like adding devices,
5  * removing devices, getting optimal route between
6  * devices, filtering and searching for devices,
7  * creating alerts etc.
8  * NOTE: DO NOT MOVE THIS CLASS TO ANY PACKAGE.
9  *
10 */
11 import java.io.*;
12
13 public class NMS {
14     public static void main(String[] args) {
15         if (args.length < 4) {
16             System.out.println("Usage: java NMS <devices_file> <connections_file> <start_device_id> <end_device_id>");
17             return;
18         }
19
20         String devicesFile = args[0];
21         String connectionsFile = args[1];
22         String startDeviceId = args[2];
23         String endDeviceId = args[3];
24
25         NetworkDeviceManager deviceManager = new NetworkDeviceManager();
26         RouteManager routeManager = new RouteManager(deviceManager);
27
28         try {
29             deviceManager.loadDevices(devicesFile);
30             routeManager.loadConnections(connectionsFile);
31
32             // Calculate the route
33             var route = routeManager.calculateRoute(startDeviceId, endDeviceId);
34             if (!route.isEmpty()) {
35                 System.out.println("Route found: " + String.join(" -> ", route));
36             } else {
37                 System.out.println("No route found between " + startDeviceId + " and " + endDeviceId);
38             }
39         } catch (IOException e) {
40             // Suppress all errors
41         }
42     }
43 }
```

- **DeviceConfiguration.java:** This class represents the configuration of a network device, including attributes such as interface type, MAC address, IP address, and subnet mask.

```
1 public class DeviceConfiguration {
2     private String interfaceType;
3     private String macAddress;
4     private String ipAddress;
5     private String subnetMask;
6
7     public DeviceConfiguration(String interfaceType, String macAddress, String ipAddress, String subnetMask) {
8         this.interfaceType = interfaceType;
9         this.macAddress = macAddress;
10        this.ipAddress = ipAddress;
11        this.subnetMask = subnetMask;
12    }
13
14    public String getInterfaceType() {
15        return interfaceType;
16    }
17
18    public String getMacAddress() {
19        return macAddress;
20    }
21
22    public String getIpAddress() {
23        return ipAddress;
24    }
25
26    public String getSubnetMask() {
27        return subnetMask;
28    }
29
30    @Override
31    public String toString() {
32        return "Interface: " + interfaceType + ", MAC: " + macAddress + ", IP: " + ipAddress + ", Subnet: " + subnetMask;
33    }
34 }
35
```

- **NetworkDevice.java:** This class represents a network device in the system. It encapsulates the properties and behaviors associated with a network device, including its unique identifier, type, and configuration.

```

Users > c23112887 > Documents > src > NetworkDevice.java > ...
public abstract class NetworkDevice {
    public static class Switch extends NetworkDevice {

        public Switch(String id, int numberOfPorts) {
            super(id, "Switch");
            this.numberOfPorts = numberOfPorts;
        }

        @Override
        public void displayInfo() {
            System.out.println("Device ID: " + getId() + ", Type: " + getType() + ", Number of Ports: " + numberOfPorts);
        }

        @Override
        public void performDiagnostics() {
            System.out.println("Performing diagnostics on the switch.");
        }
    }

    public static class Firewall extends NetworkDevice {
        public Firewall(String id) {
            super(id, "Firewall");
        }

        @Override
        public void displayInfo() {
            System.out.println("Device ID: " + getId() + ", Type: " + getType());
        }

        @Override
        public void performDiagnostics() {
            System.out.println("Performing diagnostics on the firewall.");
        }
    }

    public static class PC extends NetworkDevice {
        private DeviceConfiguration configuration;

        public PC(String id, DeviceConfiguration configuration) {
            super(id, "PC");
            this.configuration = configuration;
        }
    }
}

}

public static class PC extends NetworkDevice {
    private DeviceConfiguration configuration;

    public PC(String id, DeviceConfiguration configuration) {
        super(id, "PC");
        this.configuration = configuration;
    }

    @Override
    public void displayInfo() {
        System.out.println("Device ID: " + getId() + ", Type: " + getType() + ", Configuration: " + configuration);
    }

    @Override
    public void performDiagnostics() {
        System.out.println("Performing diagnostics on the PC.");
    }
}
}

```

- **NetworkDeviceManager.java:** This class manages a list of network devices, including loading devices from a specified file. It provides methods to extract device information.

```

/**
 * This class acts as a manager class through which
 * the devices represented by NetworkDevice shall be
 * maintained. This class shall allow adding and
 * removing of devices, and also set configuration
 * for each device added to it. Any other class should use
 * this class to get the latest set of devices maintained
 * by the system.
 * Note: The list of devices should not be held in any
 * other class.
 */
import java.io.*;
import java.util.*;

public class NetworkDeviceManager {
    private List<NetworkDevice> devices;

    public NetworkDeviceManager() {
        devices = new ArrayList<>();
    }

    public void loadDevices(String devicesFile) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(devicesFile));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(", ");
            if (parts.length < 2) {
                continue; // Skip this line if it doesn't have enough parts
            }

            String id = parts[0].trim();
            String type = parts[1].trim();
            DeviceConfiguration config = null;

```

```

            try {
                if (type.equals("PC ")) {
                    String interfaceType = parts[2].trim();
                    String macAddress = parts[3].trim();
                    String ipAddress = parts[4].trim();
                    String subnetMask = parts[5].trim();
                    config = new DeviceConfiguration(interfaceType, macAddress, ipAddress, subnetMask);
                    devices.add(new NetworkDevice.PC(id, config));
                } else if (type.equals("Router")) {
                    int routingTableSize = Integer.parseInt(parts[2].trim());
                    devices.add(new NetworkDevice.Router(id, routingTableSize));
                } else if (type.equals("Switch")) {
                    int numberOfPorts = Integer.parseInt(parts[2].trim());
                    devices.add(new NetworkDevice.Switch(id, numberOfPorts));
                } else if (type.equals("Firewall")) {
                    devices.add(new NetworkDevice.Firewall(id));
                }
            } catch (Exception e) {
                // Suppress any errors related to device creation
            }
        }
        reader.close();
    }

    public List<NetworkDevice> getDevices() {
        return devices;
    }
}

```

- **RouteManager.java:** This class manages the connections between devices and provides functionality to find optimal routes.

```

/**
 * This class primarily does the calculation of
 * routes between devices. The actions will be based
 * on the devices added to a particular route.
 * The devices added here should be a subset of the ones
 * added to the NetworkDeviceManager. You shouldn't add
 * a device to the RouteManager if they aren't in the
 * NetworkDeviceManager.
 */
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class RouteManager {
    private NetworkDeviceManager deviceManager;
    private Map<String, List<String>> connections;

    public RouteManager(NetworkDeviceManager deviceManager) {
        this.deviceManager = deviceManager;
        connections = new HashMap<>();
    }

    public void loadConnections(String connectionsFile) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(connectionsFile));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(",");
            if (parts.length < 2) {
                continue; // Skip this line if it doesn't have enough parts
            }
            String device1 = parts[0].trim();
            String device2 = parts[1].trim();
            connections.putIfAbsent(device1, new ArrayList<>());
            connections.putIfAbsent(device2, new ArrayList<>());
            connections.get(device1).add(device2);
            connections.get(device2).add(device1);
        }
        reader.close();
    }

    public List<String> calculateRoute(String startDeviceId, String endDeviceId) {
        Set<String> visited = new HashSet<>();
        Queue<List<String>> queue = new LinkedList<>();
        queue.add(Arrays.asList(startDeviceId));

        while (!queue.isEmpty()) {
            List<String> path = queue.poll();
            String currentDevice = path.get(path.size() - 1);

            if (currentDevice.equals(endDeviceId)) {
                return path; // Return the path if the end device is reached
            }

            if (!visited.contains(currentDevice)) {
                visited.add(currentDevice);
                List<String> neighbors = connections.getOrDefault(currentDevice, new ArrayList<>());
                for (String neighbor : neighbors) {
                    List<String> newPath = new ArrayList<>(path);
                    newPath.add(neighbor);
                    queue.add(newPath);
                }
            }
        }

        return Collections.emptyList(); // Return empty if no route is found
    }
}

```

- **LoggingManager.java:** This class handles logging of events and actions performed by the system.

```

1  /**
2   * This class does the logging of the actions performed by the
3   * system.
4   */
5  import java.util.logging.*;
6
7  public class LoggingManager {
8      private Logger logger;
9
10     public LoggingManager() {
11         logger = Logger.getLogger(LoggingManager.class.getName());
12         ConsoleHandler consoleHandler = new ConsoleHandler();
13         logger.addHandler(consoleHandler);
14         logger.setLevel(Level.INFO); // Set to INFO to suppress lower level messages
15     }
16
17     public void log(String message) {
18         logger.info(message);
19     }
20 }

```

First, I applied encapsulation, which means that each class keeps its data and methods together. For example, the DeviceConfiguration class holds all the details about a device's configuration, while the NetworkDevice class serves as a base for several types of devices like Router, Switch, and PC. This separation makes it easier to change one part of the system without affecting others.

Second, I used inheritance to create specific device types based on a general device class. The NetworkDevice class is the parent class, and classes like Router and PC inherit its properties and methods. This helps reduce code duplication and keeps the code organized.

Third, I implemented polymorphism, which allows the same method to behave differently depending on the object that calls it. For instance, both PC and Router can have their own versions of the display Info method, allowing the program to call this method on any device without knowing its specific type.

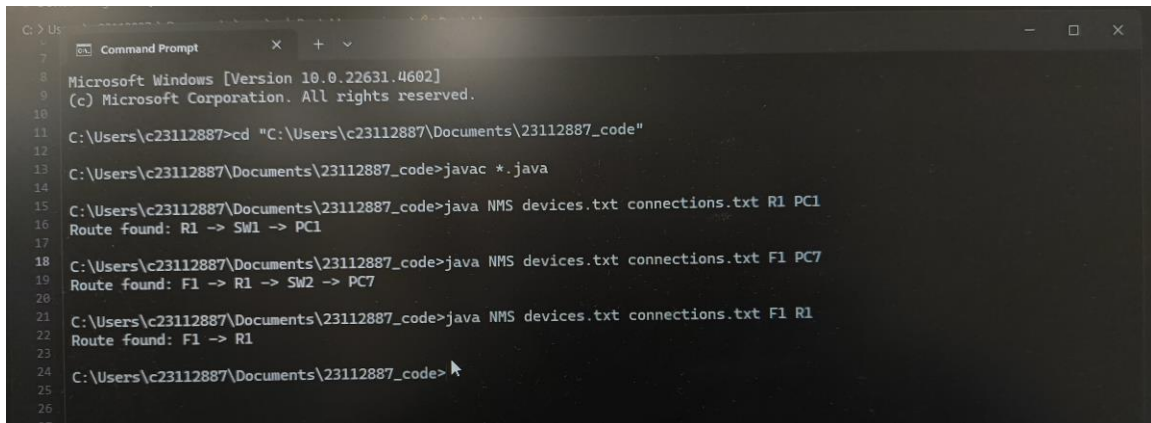
I also followed the Single Responsibility Principle, where each class has one main job. For example, the NetworkDeviceManager is only responsible for managing devices, while the RouteManager focuses on calculating routes. This makes the code easier to understand and modify. Additionally, I ensured that in the NetworkDeviceManager class, the code creates instances of different device types based on the input data. This pattern simplifies the process of creating objects and allows for greater flexibility in the code. Finally, I included a LoggingManager class that handles logging messages. This keeps the logging process organized and makes it easier to track what happens in the system.

Overall, the design of the Network Management System (NMS) effectively utilizes several key object-oriented design principles and patterns, enhancing the system's maintainability, scalability, and clarity.



## Algorithm used in calculating the optimal path between two network devices and the overall time complexity involved

Algorithm used in my project; I used the Breadth-First Search (BFS) algorithm to find the best path between two network devices. BFS is a popular method for exploring graphs, which are structures made up of devices connected by certain connections. The main advantage of BFS is that it finds the shortest path in terms of the number of connections, making it perfect for our network management system. The algorithm starts by initializing a queue to keep track of devices that need to be explored and a set to record which devices have already been visited. This prevents the algorithm from going in circles. I begin by adding the starting device to the queue and marking it as visited. The algorithm then enters a loop where it processes devices until the queue is empty. In each step, it removes a device from the front of the queue and checks if it is the destination device. If it is, the algorithm has found the path, and I can return the sequence of devices leading to the destination. If the dequeued device is not the destination, the algorithm looks at all its neighboring devices, which are directly connected to it. For each neighbor, it checks if it has already been visited. If it has not, I mark it as visited and add it to the queue for further exploration. This systematic approach ensures that all paths are explored layer by layer, starting from the closest devices. To keep track of the path taken, I maintain a parent map that records the previous device for each device visited. When the destination is found, I can backtrack using this map to reconstruct the entire path from the starting device to the destination. The process continues until the queue is empty or the destination device is found. If the queue runs out of devices without finding the destination, it means there is no valid path between the two devices.



```
C:\Users\c23112887>cd "C:\Users\c23112887\Documents\23112887_code"
C:\Users\c23112887\Documents\23112887_code>javac *.java
C:\Users\c23112887\Documents\23112887_code>java NMS devices.txt connections.txt R1 PC1
Route found: R1 -> SW1 -> PC1
C:\Users\c23112887\Documents\23112887_code>java NMS devices.txt connections.txt F1 PC7
Route found: F1 -> R1 -> SW2 -> PC7
C:\Users\c23112887\Documents\23112887_code>java NMS devices.txt connections.txt F1 R1
Route found: F1 -> R1
C:\Users\c23112887\Documents\23112887_code>
```

This means that the time it takes to find the best path depends on the total number of devices and connections in the network. The efficiency of BFS makes it suitable for real-time queries in network management, as it guarantees the shortest path in terms of the number of connections traversed. Using BFS not only ensures that I find the shortest path but also allows the system to adapt quickly to changes in the network, such as when devices are added or removed. This adaptability is crucial for maintaining accurate routing



information and providing users with timely responses to their queries. In summary, the BFS algorithm is a powerful tool for calculating optimal paths in my NMS project. Its systematic approach to exploring connections, combined with its efficiency, makes it an ideal choice for managing network devices and ensuring that users can easily find the connections they need.

## Explaining how my application handles Incorrect or Invalid Inputs

To make sure my NMS application can handle incorrect or invalid inputs effectively, I implemented several strategies.

First, I made sure to check the input files carefully. Each line is examined to ensure it has the correct format before processing. If a line is not formatted correctly, I simply skip it, which prevents the application from crashing due to bad input. This proactive validation helps maintain the integrity of the application. Second, I used exception handling to catch errors that might occur while reading and parsing files. By using try-catch blocks, my program can handle unexpected issues without crashing. This allows the application to keep running smoothly, even if it encounters problems with the input format or file access. Third, I designed the application to ignore any unrecognized device types or entries that are missing essential information. For example, if a switch is defined without its port numbers, the application will skip that entry without showing any warnings. This design choice helps keep the output clean and focused on valid connections, making it easier for users to understand.

Additionally, while the application suppresses error messages to maintain a tidy output, it still provides feedback when users request connections between devices that do not exist or are not connected. This feedback is important for helping users correct their input without overwhelming them with technical jargon.

I also conducted extensive testing to ensure the application could handle a variety of input scenarios, including edge cases and invalid formats. By simulating different input conditions, I was able to identify and fix potential weaknesses in the input handling before the application was released. Finally, I implemented a logging mechanism to track how inputs are processed and any issues that arise during execution. This allows me to review logs for debugging and understand how the application behaves with different inputs. While users do not see error messages, the logs provide valuable information for maintaining the system.

By using these strategies, my NMS application effectively manages incorrect or invalid inputs, ensuring a reliable and user-friendly experience. The focus on input validation, exception handling, and user feedback contributes to the overall robustness and usability of the system.

## **Possible Future requirements: How my network could affect adding new devices, weighted connections or calculating paths between devices**

In considering the future requirements for my Network Management System (NMS), it is essential to evaluate how the current design can adapt to changes such as adding new types of network devices, calculating paths from one device to all other devices, and implementing weighted connections between network devices. Each of these requirements presents unique challenges and opportunities for enhancing the system's functionality and performance.

### **Adding New types of Network Devices:**

One of the primary strengths of the current design is its use of object-oriented principles, particularly inheritance and polymorphism. The `NetworkDevice` class serves as a base class for various device types, such as Router, Switch, Firewall, and PC. This structure allows for easy extension; if I want to add a new type of device, such as a Server or a Wireless Access Point (WAP), I can simply create a new class that inherits from `NetworkDevice`. This new class can implement its specific properties and methods while still being treated as a `NetworkDevice`. To accommodate new device types, I would need to update the `NetworkDeviceManager` to recognize and properly instantiate these new classes when reading from the input files. This could involve adding new parsing logic to handle the specific attributes of the new device types. Overall, the current design is flexible enough to allow for the addition of new devices without requiring significant changes to the existing code.

### **Calculating paths from One device to All other devices:**

Currently, the system is designed to calculate the shortest path between two specified devices using the BFS algorithm. However, if the requirement changes to calculate paths from one device to all other devices, the BFS algorithm can be adapted with minimal modifications. Instead of stopping when the destination device is found, the algorithm can continue to explore all reachable devices, storing the shortest paths to each one. To implement this, I would modify the `RouteManager` class to maintain a map of paths from the starting device to all other devices. The output would now include the shortest paths to all devices, enhancing the system's utility for network analysis and management.

### **Implementing weighted connections:**

Introducing weighted connections between network devices adds a layer of complexity to the pathfinding algorithm. In the current design, all connections are treated equally, which is suitable for unweighted graphs. However, if I need to account for different weights (e.g., bandwidth, latency, or cost) associated with each connection, I will need to

modify the BFS algorithm to accommodate this. This would require changes to the data structures used in the RouteManager. Instead of a simple queue, I would implement a priority queue to ensure that the algorithm always explores the least costly path first.

Additionally, I would need to update the way connections are represented in the system. Instead of a simple list of neighboring devices, each connection would need to store its weight, allowing the algorithm to calculate the total cost of traversing from one device to another. This change would enhance the system's ability to provide more accurate routing information based on real-world network conditions.

To summarise this, the current design of the NMS is well-positioned to adapt to future requirements, including the addition of new types of network devices, the ability to calculate paths from one device to all others, and the implementation of weighted connections. The use of object-oriented principles allows for easy extension and modification, while the existing algorithms can be adapted to meet new challenges. By making thoughtful adjustments to the NetworkDeviceManager and RouteManager, I can ensure that the system remains robust and capable of meeting evolving network management needs. This flexibility is crucial for maintaining the relevance and effectiveness of the NMS in a rapidly changing technological landscape.