Smart Health Monitoring System - Java Project (Detailed Guide)

1. Core Feature Implementation

The Smart Health Monitoring System is designed to monitor three primary health metrics:

- Body Temperature
- Blood Pressure
- Heart Rate

Each metric is input via a user-friendly GUI interface. The system checks whether each value is within a healthy range and provides real-time feedback. It also prepares a health summary report and supports modular extension.

Key objectives:

- Collect health data from users
- Perform real-time health analysis
- Provide feedback and suggestions
- Allow for future expansion (e.g., wearable integration, doctor alerts)

Java GUI Implementation (MainWindow.java)

```java
import javax.swing.*;
import java.awt.event.*;

public class MainWindow extends JFrame {
    private JTextField tempField, bpField, hrField;
    private JButton submitButton, clearButton;

    public MainWindow() {
        setTitle("Smart Health Monitoring System");
        setSize(450, 350);
        setLayout(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JLabel title = new JLabel("Enter Your Vital Information:");
        title.setBounds(120, 10, 250, 25);
        add(title);
        JLabel tempLabel = new JLabel("Temperature (°F):");
        tempLabel.setBounds(50, 60, 150, 25);
        add(tempLabel);
        tempField = new JTextField();
        tempField.setBounds(200, 60, 150, 25);
        add(tempField);

        JLabel bpLabel = new JLabel("Blood Pressure:");
        bpLabel.setBounds(50, 100, 150, 25);
        add(bpLabel);
        bpField = new JTextField();
        bpField.setBounds(200, 100, 150, 25);
        add(bpField);

        JLabel hrLabel = new JLabel("Heart Rate:");
        hrLabel.setBounds(50, 140, 150, 25);
        add(hrLabel);
        hrField = new JTextField();
        hrField.setBounds(200, 140, 150, 25);
        add(hrField);

        submitButton = new JButton("Submit");
        submitButton.setBounds(80, 200, 120, 30);
        add(submitButton);

        clearButton = new JButton("Clear");
        clearButton.setBounds(220, 200, 120, 30);
        add(clearButton);

        submitButton.addActionListener(e -> analyzeHealth());
        clearButton.addActionListener(e -> clearFields());

        setVisible(true);
    }

    private void analyzeHealth() {
        try {
            double temp = Double.parseDouble(tempField.getText());
            int bp = Integer.parseInt(bpField.getText());
            int hr = Integer.parseInt(hrField.getText());
            if (!Validator.isValidTemperature(temp) ||
                !Validator.isValidBP(bp) ||
                !Validator.isValidHeartRate(hr)) {
                    JOptionPane.showMessageDialog(this, "Entered values are outside acceptable
range.");
                return;
            }

            String result = HealthAnalyzer.analyze(temp, bp, hr);
            JOptionPane.showMessageDialog(this, result);
            HealthReport.saveReport(result);

        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "Invalid input! Please enter only numbers.");
        }
    }

    private void clearFields() {
        tempField.setText("");
        bpField.setText("");
        hrField.setText("");
    }

    public static void main(String[] args) {
        new MainWindow();
    }
}
```

2. Error Handling and Robustness

The system uses Java's try-catch mechanism to handle incorrect or malformed inputs (e.g., entering letters instead of numbers). It uses safe default behaviors, informative error dialogs, and boundary checks to make the system robust and prevent crashes. The 'Validator' class encapsulates input range checking to keep logic reusable and organized.

Validator.java - Range Validation Logic

```
public class Validator {
    public static boolean isValidTemperature(double temp) {
        return temp >= 90 && temp <= 110;
    }
    public static boolean isValidBP(int bp) {
        return bp >= 80 && bp <= 200;
    }
    public static boolean isValidHeartRate(int hr) {
        return hr >= 40 && hr <= 180;
    }
}
```

3. Integration of Components

The system's GUI, backend analysis logic, and file logging components are modular but work cohesively:

- GUI collects data and sends to backend
- Backend returns evaluation
- Evaluation is displayed to user and optionally saved

This clear separation supports code reusability and future extensibility (e.g., connecting to databases or APIs).

4. Event Handling and Processing

```
// Part of GUI code - Button click event handling
submitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        analyzeHealth();
    }
});

clearButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        clearFields();
    }
```

});

5. Data Validation

Validation ensures that only appropriate numeric values within defined ranges are accepted. This prevents logical errors in the health analysis and ensures the integrity of health reports. Each metric has defined acceptable bounds, and values outside this range are flagged as invalid.

6. Health Analysis Logic (HealthAnalyzer.java)

```java
public class HealthAnalyzer {

    public static String analyze(double temp, int bp, int hr) {
        StringBuilder result = new StringBuilder("Health Summary:\n");

        if (temp < 97 || temp > 99)
            result.append("- Temperature Alert: ").append(temp).append("°F\n");
        else
            result.append("- Temperature Normal\n");

        if (bp < 90 || bp > 140)
            result.append("- Blood Pressure Alert: ").append(bp).append("\n");
        else
            result.append("- Blood Pressure Normal\n");

        if (hr < 60 || hr > 100)
            result.append("- Heart Rate Alert: ").append(hr).append("\n");
        else
            result.append("- Heart Rate Normal\n");

        return result.toString();
    }
}
```

7. Health Report Generation (HealthReport.java)

```java
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class HealthReport {
    public static void saveReport(String data) {
        try {
            FileWriter writer = new FileWriter("health_report.txt", true);
             String timestamp = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
            writer.write("[" + timestamp + "]\n" + data + "\n\n");
            writer.close();
        } catch (IOException e) {
            System.out.println("Error saving report.");
        }
    }
}
```

8. Project Structure & Documentation

Folder Layout:

```
/SmartHealthMonitor/
|-- src/
|   |-- MainWindow.java
|   |-- Validator.java
|   |-- HealthAnalyzer.java
|   |-- HealthReport.java
|-- README.md
|-- health_report.txt
```

README.md Contents:

- Overview of the project
- Setup and usage instructions
- List of features
- How to compile and run

Smart Health Monitoring System - Java Project (Detailed Guide)

This structure keeps the codebase clean and easy to navigate.