Mood Tracker Application Report

Introduction

The **Mood Tracker Application** is a console-based Java application designed to help users track their moods by creating, managing, and analysing mood entries. Built using a **layered architecture**, the application ensures separation of concerns and maintainability. It uses modern Java features and best practices.

The application allows users to:

- Add, edit, and delete mood entries.
- View mood history and filter entries based on intensity or date.
- Group, partition, and sort mood entries.
- Display unique mood descriptions and analyse mood statistics.

The project demonstrates a wide range of Java features, including object-oriented principles, functional programming, and advanced Stream API operations.

User Stories

1. **As a developer, I want to use Java Streams so that I can efficiently filter, group, and transform mood entries.**
2. **As a developer, I want to use the Optional class so that I can handle null values safely when retrieving mood entries.**
3. **As a developer, I want to use lambda expressions so that I can perform concise operations on collections.**
4. **As a developer, I want to use method references so that I can improve code readability when processing mood entries.**
5. **As a developer, I want to use the Comparator interface so that I can sort mood entries by date and intensity.**
6. **As a developer, I want to use Java's functional programming features so that I can write cleaner and more maintainable code.**
7. **As a developer, I want to use the Map and Collectors API so that I can efficiently group and count mood entries.**
8. **As a developer, I want to use the List interface and its utility methods so that I can easily manipulate mood entries.**
9. **As a developer, I want to use Java’s Predicate interface so that I can define filtering conditions for mood entries.**
10. **As a developer, I want to use Streams’ reduce method so that I can compute aggregated results like the most intense mood.**
11. **As a developer, I want to use exception handling with try-catch blocks so that I can manage errors gracefully in the service.**
12. **As a developer, I want to use unit tests with JUnit so that I can verify the correctness of my mood service methods.**

Evaluation

Adherence to the Project Brief

The application adheres well to the project brief by demonstrating a wide range of Java features and implementing them in a practical way. The layered architecture ensures separation of concerns, making the application maintainable and scalable. The use of modern Java features like lambdas, method references, and Stream API operations enhances the readability and efficiency of the code.

The application also demonstrates error handling through custom exceptions. For example:

- Custom exceptions like `MoodEntryNotFoundException` provide meaningful error messages.
- Defensive copying is used to protect mutable data structures.

The project successfully integrates advanced Java features, such as:

- Functional programming constructs (`Predicate`, `Stream API`).
- Modern Java constructs like switch expressions and records.
- Stream API operations like `groupingBy`, `partitioningBy`, and `toMap`.

Problems Encountered

1. **Database Integration:**
    - Initially, the application used an H2 in-memory database, but it was later switched to MySQL for persistent storage. This required additional configuration to ensure compatibility.
2. **Stream API Complexity:**
    - While implementing advanced Stream API operations, ensuring proper handling of null values and collections was challenging.
3. **Error Handling:**
    - Designing meaningful custom exceptions and ensuring they were used consistently across the application required careful consideration.
4. **Balancing Features and Simplicity:**
    - While the application demonstrates many Java features, balancing feature with simplicity was a challenge. Efforts were made to keep the codebase clean and maintainable.

Strengths

1. Demonstrates a wide range of Java features, from object-oriented principles to modern constructs.
2. Uses a layered architecture for better separation of concerns.
3. Implements good practice for error handling and defensive coding practices.
4. Leverages the Stream API for efficient data processing.

Areas for Improvement

- **User Interface:** The console-based interface could be replaced with a web or desktop GUI for better usability.
- **Testing:** More comprehensive unit tests could be added to ensure code reliability.