Conception Phase: Habit Tracking App

Course: Object Oriented and Functional Programming with Python (DLBDSOOFPP01)

Name: **<Kiran Vani Himaja Sarma Ravada>** MatrNo: **<Matriculation Number: UPS10796533>**

1 Objective and Scope

The goal of this project is to design and implement a Python-based backend for a habit tracking application. The system focuses on the essential functionality: users can define habits with a periodicity (daily or weekly), check off completions, persist data between sessions, and analyse progress such as streaks. The solution uses **object-oriented programming** for the domain model and core operations, and **functional programming** for analytics functions. A **command-line interface (CLI)** will serve as the primary user interface.

2 System Overview and Architecture

The application is designed as a small modular system with clear responsibilities:

- **CLI Layer**: Parses user commands (e.g., add habit, check off, list, analyse) and prints results.
- **Domain Model**: Encodes the concept of a habit and its task completions using classes.
- **Repository (Persistence Layer)**: Stores and retrieves habits and completion events from persistent storage.
- **Analytics Module**: Provides pure functions to compute reports (lists, filters, and streak calculations).

The overall flow is: *User* $\rightarrow$ *CLI* $\rightarrow$ *Repository* $\leftrightarrow$ *SQLite Database*, and for analysis: *CLI* $\rightarrow$ *Analytics* (using data loaded via the repository).

3 Data Model and Core Classes

3.1 Habit

A habit represents a clearly defined task to be completed periodically. The `Habit` class stores:

- `id` (unique identifier)
- `name` (human-readable name)
- `task` (task specification)
- `periodicity` (`daily` or `weekly`)
- `created_at` (timestamp when the habit was created)

3.2 Completion

A completion represents a single check-off event for a habit at a specific timestamp. It stores:

- `habit_id` (references a habit)

- `completed_at` (timestamp of completion)

The relationship is **one habit to many completions**.

4 Streak Definition and Period Logic

A habit is considered *completed for a period* if it has at least one completion timestamp within that period. A period is:

- **Daily**: one calendar day window
- **Weekly**: one calendar week window (e.g., ISO week)

A **streak** is defined as the number of **consecutive periods** in which the habit was completed at least once. If a user misses a period, the streak is broken. The system will compute:

- longest streak per habit
- longest streak across all habits

5 Persistence Strategy

To persist data between sessions, the project uses **SQLite** via Python's built-in `sqlite3` module. SQLite is a lightweight relational database well-suited for this scope, provides data integrity, and keeps the deployment simple. Two tables are sufficient:

- `habits(id, name, task, periodicity, created_at)`
- `completions(id, habit_id, completed_at)` with a foreign key to `habits`

6 Analytics Module (Functional Programming)

The analytics layer is implemented using a functional approach: small, composable, and side-effect-free functions that operate on collections of `Habit` and `Completion` objects. Planned analytics functions include:

- return a list of all currently tracked habits
- return a list of all habits with the same periodicity (daily/weekly)
- return the longest streak of all habits
- return the longest streak for a given habit

Implementation will use common functional patterns such as `map`, `filter`, and reductions over computed period windows.

7 CLI Design (Clean User API)

A command-based CLI will be provided to ensure a clear and testable API. Example commands:

- `habits add -name "Workout" -task "20 min" -period daily`
- `habits check -name "Workout"`

- `habits list -period daily`
- `habits analyze -longest`
- `habits analyze -habit "Workout"`

This interface keeps user interaction simple and matches the required create/manage/analyse workflow.

8 UML Diagram (Components and Relationships)

Figure 1 shows the planned structure and interaction of components.

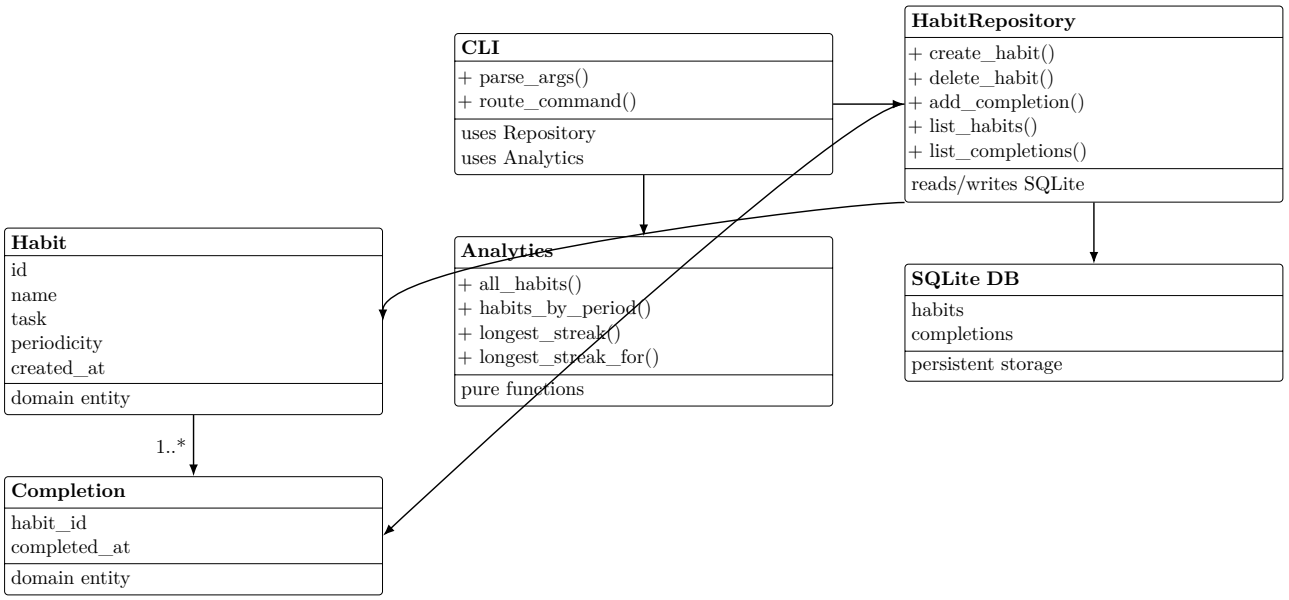

Figure 1: Planned class/components diagram for the habit tracking app.

9 Testing Strategy (Brief)

Critical logic (period completion and streak calculation) will be validated using unit tests (e.g., `pytest`). Tests will cover: correct streak computation, daily vs weekly period windows, repository read/write behaviour, and analytics outputs.

Note: In the development phase, the design will be refined based on implementation feedback, while keeping the above architecture stable.