Conceptual phase

1. Introduction

This project aims to build a robust and comprehensive Habit Tracking Application based on Object–Oriented and Functional Programming with the Python language. The system will help users define and track habits that they define from the list of available habits, such as reading for 10 minutes, going to the gym, drawing, etc. The application will be built emphasizing Object–Oriented Design (OOD) and functional programming techniques (FP).

The application will allow the users to:

- Create and mark down the task, either daily or periodically;
- Track the "streaks," consecutive number of done habits;
- Analyze the progress and, if needed, provide a summary of the longest streak;

2. Object – Oriented Design

Five main core components:

- Habit
  - Attributes: id, name, periodicity, created_at
  - Methods: add_completion(), get_current_streak(), get_longest_streak(), is_broken()
- HabitTracker
  - Holds a collection of Habit instances
  - Methods: create_habit(), delete_habit(), list_habits(), mark_complete()
- AnalyticsModule
  - Functions: No classes: calculate_longest_streaks(habits), filter_broken(habits), count_by_periodicity(habits)
  - Uses functional techniques—map, filter, reduce—to process habit data efficiently
- StorageHandler
  - Manages SQLite persistence: initializes tables (habits, completions), INSERT/SELECT operations for habits and completions
- CLI interface
  - A simple input loop with menu options: create, log, view habits, run analytics, exit.
  - Delegates actions to HabitTracker, AnalyticsModule, and StorageHandler

The core of everything is the Habit class, and every habit will have a name, a period ("daily" or "weekly"), a creation timestamp, and a list of completions. Around the Habit class, there will be a HabitTracker class, which acts as a journal. It will keep the habits in one place, help add new ones, delete them, and keep records. The HabitTracker class acts as a library of habits with a helpful librarian keeping everything organized. Moreover, there is AnalyticsModule(functional), which will act as the brain of the application; using functional programming, this module will calculate things

like "what is the longest streak?", "Which habit did the user break the most?" etc. Nevertheless, the data has to be stored somewhere, such as SQLite, which is built into Python. SQLite gives the application an actual mini-database to store habits and completions. The StorageHandler will handle everything behind the scenes: creating tables, saving new records, and loading everything when the application starts. In order to interact with everything, the application will be built using a simple input() main loop through the Python built-in function. When the application runs, it will greet with the menu "Create a Habit," "Mark habit as completed, ""Show analytics," etc.

3. The Design of Data and User Flow

When we imagine someone using the app, we do not picture a software developer experienced in applications and how they work; we picture a simple human who may be tired and trying to gain new habits. Maybe they are trying to read every day or get into journaling every morning; the last thing the user needs is a confusing application.

User Flow:

1. The user launches the app in a terminal.
2. The CLI shows options to create a habit, mark its completion, view habits, run analytics, or exit.
3. Choosing "Create habit" prompts for a name, whether daily or weekly, and the app adds it to SQLite.
4. When users mark a habit "done," the app logs a timestamp in the completions table.
5. If the user asks for analytics, the AnalyticsModule runs SQL queries through StorageHandler, builds a list of habits, and uses pure functions to calculate streaks, breaks, and habit types.
6. The user sees a simple summary like "Daily habit 'Meditate' – current streak: 5 days; longest: 8."

Data Flow (technical):

- Creation → CLI → HabitTracker.create_habit() → StorageHandler.insert_habit() → DB
- Completion → CLI → HabitTracker.mark_complete() → StorageHandler.insert_completion()
- Analytics → CLI → StorageHandler.load_habits_with_completions() → list of Habit objects → AnalyticsModule processes → results to CLI

4. Draft of Class Structure

At the center of design is the idea that each class should do one thing. The project has split the responsibilities across clearly defined classes and modules.

Main classes:

Habit: It carries the basic information:

- id: an integer, primary key in the database

- name: what the user is trying to build (e.g., "Workout," "Read")
- periodicity: either 'daily' or 'weekly'
- created_at: timestamp for when it was born

The methods:

- add_completion(date)
- get_current_streak()
- get_longest_streak()
- was_broken()

HabitTracker: This class acts as a bridge between the user interface and the logic. It handles:

- Creating habits
- Storing and retrieving them
- Assigning tracking to StorageHandler

StorageHandler: This is the memory system. It wraps SQLite queries:

- create_tables()
- insert_habit()
- insert_completion()
- fetch_all_habits()
- fetch_completions()

AnalyticsModule: It takes habit data and returns answers like:

- What is the longest streak?
- Which habit has been broken the most?
- How many habits are daily vs weekly?

5. Explanation of the Role of Functional Programming

Pure functions are those that return the same result given the same input. A streak counter takes a list of dates and gives a number.

- filter() → to select only completed habits by periodicity
- map() → to transform completion dates into intervals
- reduce() → to calculate maximum streaks across all habits

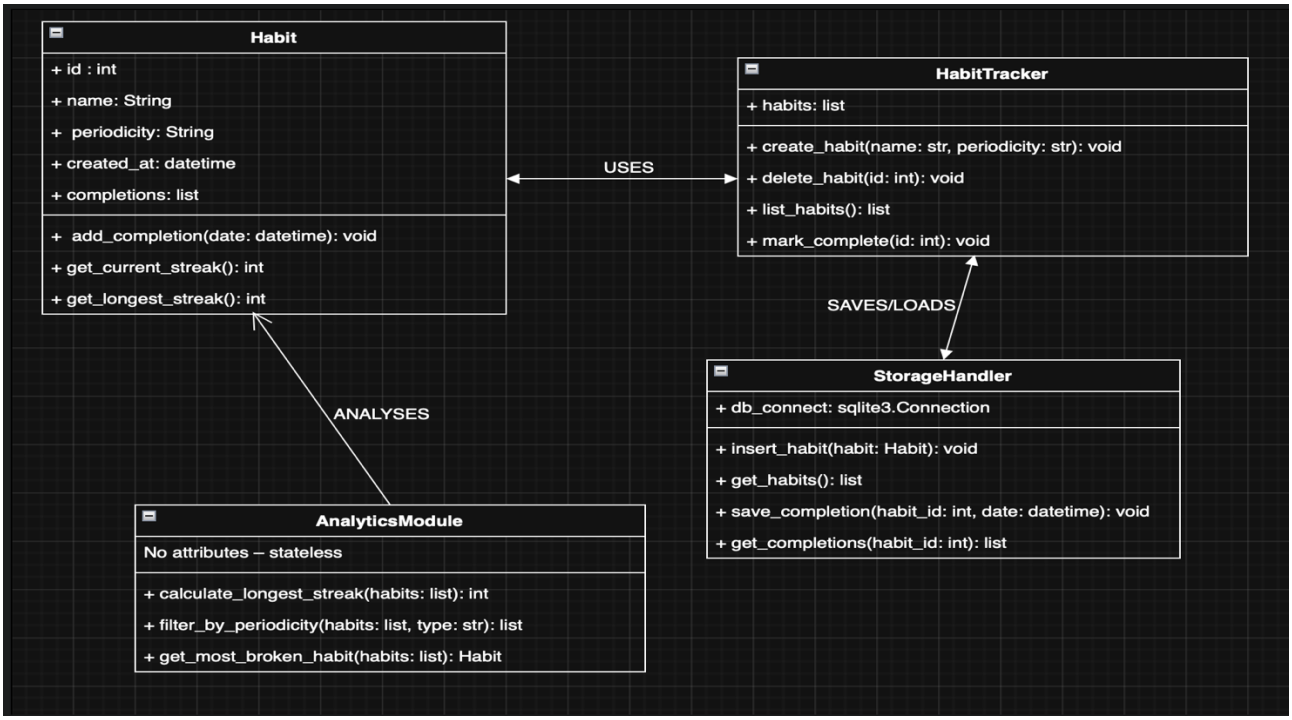
6. Justification of Technology Choices

The technologies used for this project were chosen mainly because of their simplicity, transparency, and resilience. Those three goals were the main aim.

- Python 3.7+ was a natural choice.

- SQLite is more powerful and structured than flat files but does not demand a server setup like MySQL. Since it is embedded, it keeps everything local and clean.
- Command-line interface using input() might not be glamorous, but it is accessible. It keeps the user focused on function, not form. Anyone can run this from a terminal.
- Functional programming for analytics; it keeps business logic predictable, reusable, and mathematically sound.

UML Class Diagram:



UML Sequence Diagram: