Conception Phase

Habit Tracker – Habit Lifecycle

Habit Tracking Application Overview :

The goal of this application is to help users build and maintain habits by tracking their daily behavior, monitoring streaks, and analyzing long-term consistency. Instead of treating habits as simple records, the system models each habit as a behavioral entity that evolves over time .

The application is centred around a **Daily Check-in Dashboard**, which acts as the main interaction hub. From this dashboard, users can create new habits, check off completed habits, analyze their progress by streaks , edit or delete habits, and exit the system. After every operation, the system returns the user to the dashboard, creating a continuous daily interaction loop.

The application is implemented in Python and uses a **SQLite database sqlite3** for persistent storage.

System Architecture

```
habit-tracker/
├── main.py          ← Entry point (CLI + dashboard)
├── habit.py         ← Represents one habit and its behavior (core entity)
├── tracker.py       ← Coordinates user actions (controller)
├── storage.py       ← SQLite logic Handles persistence
├── analytics.py     ← Functional analytics
├── predefined.py    ← PredefinedHabitsLoader
│
├── habits.db        ← SQLite database
│
├── tests/
│   ├── test_analytics.py
│   ├── test_habit.py
│   └── test_storage.py
```

The system is divided into logical components, each with a clear responsibility.

Habit → OOP entity

Habit file has a class called **Habit** , in addition to this method **__init__(self, name, category, frequency, created_at=None):** , we gonna have 6 methods **: add_completion(self, completion_time=None)**

, is_broken(self,today=None), edit(self,answer,new_value), get_current_streak(self), get_longest_streak(self), to_dict(self)

The Habit class represents the core domain entity of the application. Each habit contains: a name, a category, a periodicity (daily or weekly), a creation timestamp, a list of completion timestamps.

It provides methods to: mark a habit as completed for a given day and, calculate the current streak , longest streak, determine if the habit is broken , edit it , then dict to convert habit to dictionary for storage and analytics .

Streaks are **not stored** directly but are derived dynamically from completion timestamps. This avoids data redundancy and ensures correctness at all times.

HabitTracker → OOP controller

- Tracker file has a class called **HabitTracker** has 6 methods , **create_habit(self, name, category, frequency):, check_off(self, habit_name): ,get_all_habits(self):, get_current_streak_for_habit(self, habit_name):, get_broken_habits(self): , get_habits_by_periodicity(self, freq): ,get_longest_streak_all(self):, get_longest_streak_for_habit(self, name):, edit_habit(self,habit_name,answer,new_value):, delete_habit(self,habit_name):**
- The HabitTracker component acts as the central controller of the system. It coordinates interactions between the user interface, the domain model, the analytics module, and the persistence layer.
- Responsibilities include: creating new habits, checking off habits. editing and deleting habits, retrieving habits for analysis
- The HabitTracker does not contain user input logic or database-specific logic. Instead, it exposes a clean API that can be called by the CLI or tested independently.

Analysis → functional module

- The analytics functionality is implemented using a functional programming approach. Analytics functions operate on collections of Habit objects and return computed results . Analytics file has those fcts : **get_all_habits(habits):** , **get_broken_habits(habits):**, **get_habits_by_periodicity(habits, frequency):**, **get_longest_streak_all(habits):**, **get_longest_streak_for_habit(habits, habit_name):** ,
- The analytics module provides functionality to: return all tracked habits, filter habits by periodicity, determine the longest streak across all habits , determine the longest streak for a specific habit
- Using functional programming for analytics improves testability and keeps analytical logic independent from storage and user interaction.

Storage → OOP database gateway

- The Storage component is responsible for persisting habit data between user sessions using an SQLite database.
- The database schema consists of: a habits table (habit metadata) , a completions table (timestamps of completed tasks)
- Each completion entry references a habit via a foreign key. This design ensures that raw factual data (timestamps) is stored, while derived values such as streaks are calculated dynamically.

- SQLite was chosen because it is lightweight, file-based, and does not require external services, making it well suited for this project.

It has class called **Storage ,** has 6 methods **add_habit(self, habit): ,get_all_habits(self): ,add_completion(self, habit_id, completed_at): , get_completions(self, habit_id): , update_habit(self, habit_id, answer, new_value): , delete_habit(self, habit_id):**

The Storage class handles all communication with the SQLite database .

It provides methods to: insert new habits, update habit records, store completion dates, delete , retrieve data for analysis . This separation ensures that database logic is isolated from business logic.

Predefined Habits

predefined.py → class called **PredefinedHabitsLoader,** has a method **load(self, storage):**

The system includes a set of predefined habits (three daily and two weekly) that are stored in the database when the application is first launched. These habits behave exactly like user-created habits and are included in all tracking and analysis features.

Testing:

tests/ → pytest , Tests will use: **PredefinedHabitsLoader** to populate the database.

- test_analytics.py ← streaks, filters, longest runs
- test_habit.py ← Habit logic : validates the brain of a single habit
- test_storage.py ← SQLite correctness: validates SQLite does not lie

---

This conception phase directly reflects the logic shown in the flowchart and provides a strong foundation for implementing the system using object-oriented programming