Habit Tracker Application - Conception Phase

- **Student Name**: Happiness Oribhunuebho
- **Student ID**: 92133053
- **Course**: DLBDSOOFPP01 - Object Oriented and Functional Programming with Python
- **Date**: 20th, Jan. 2026

---

1. Project Overview

1.1 Objective

The habit tracker application aims to help users build and maintain positive habits through systematic tracking and data-driven insights. The system will allow users to define habits with specific periodicities (daily or weekly), track their completion over time, and analyze their progress through various metrics including streaks and completion rates.

1.2 Core Requirements

- Create, update, and delete habits with defined periodicities
- Track habit completions with timestamps
- Calculate current and longest streaks for each habit
- Detect broken habits (missed periods)
- Provide analytics using functional programming paradigm
- Persist data between sessions using SQLite

---

2. Architecture Design

2.1 Object-Oriented Design

The application follows object-oriented principles with clear separation of concerns:

Habit Class (Core Entity)

- Represents a single trackable habit
- Encapsulates habit properties: name, periodicity, creation date, completions
- Provides methods for: adding completions, calculating streaks, checking broken status
- Implements data validation (e.g., periodicity must be 'daily' or 'weekly')

DatabaseManager Class (Persistence Layer)

- Handles all SQLite database operations
- Manages database schema creation and maintenance
- Provides CRUD operations for habits and completions
- Ensures data integrity through transactions
- Implements context manager protocol for safe resource handling

HabitTrackerCLI Class (Presentation Layer)

- Manages user interaction through command-line interface
- Coordinates between user input and business logic
- Handles command parsing and execution
- Provides user-friendly output formatting

2.2 Functional Programming Design

The **analytics module** is implemented using functional programming principles:

- **Pure Functions**: All analysis functions have no side effects and return consistent results for the same inputs
- **Higher-Order Functions**: Functions like `filter()`, `map()`, and `reduce()` operate on collections
- **Immutability**: Original data is never modified; new collections are returned
- **Function Composition**: Complex analyses are built by combining simpler functions

Key functional operations:

- `get_all_tracked_habits()`: Maps habit objects to names
- `filter_by_periodicity()`: Filters habits based on criteria
- `get_longest_streak_all_habits()`: Reduces habit streaks to maximum value
- `calculate_completion_rate()`: Pure calculation based on completion history

---

3. Data Persistence Strategy

3.1 Database Choice: SQLite

Rationale for SQLite over JSON:

- Relational structure allows complex queries (e.g., "find all completions in last 30 days")
- ACID compliance ensures data integrity
- Better performance for filtering and sorting operations
- Built-in Python support (no external dependencies)
- Professional approach suitable for production applications

3.2 Database Schema

Table: habits

```
+-------------+----------+-------------+
| Column      | Type     | Constraints |
+-------------+----------+-------------+
| id          | INTEGER  | PRIMARY KEY |
| name        | TEXT     | NOT NULL    |
| periodicity | TEXT     | NOT NULL    |
| created_at  | TEXT     | NOT NULL    |
+-------------+----------+-------------+
```

Table: completions

```
+--------------+----------+--------------------------+
| Column       | Type     | Constraints              |
+--------------+----------+--------------------------+
| id           | INTEGER  | PRIMARY KEY              |
| habit_id     | INTEGER  | FOREIGN KEY → habits(id) |
| completed_at | TEXT     | NOT NULL                 |
+--------------+----------+--------------------------+
```

Design Benefits:

- Normalized structure prevents data duplication
- Foreign key relationship maintains referential integrity
- Separate completions table allows efficient time-series queries
- Index on `(habit_id, completed_at)` optimizes streak calculations

4. User Interaction Flow

4.1 Command-Line Interface (CLI)

The application uses a simple command-based CLI:

Main Menu Loop:

1. Display prompt: `>`
2. Accept user command
3. Parse and execute command
4. Display results
5. Return to prompt

Available Commands:

- `create` → Interactive habit creation wizard
- `list` → Display all habits with status
- `complete` → Mark habit as done
- `analyze` → Access analytics submenu
- `summary` → View statistics dashboard
- `update` → Modify existing habit
- `delete` → Remove habit
- `help` → Show command list
- `exit` → Close application

4.2 User Flow Examples

Creating a Habit:

```
User: create
System: Enter habit name:
User: Exercise
System: Select periodicity (1=daily, 2=weekly):
User: 1
System: ✅ Habit 'Exercise' (daily) created!
```

Completing a Habit:

```
User: complete
System: [Shows habit list]
User: 1
System: ✅ Habit 'Exercise' marked complete!
        🔥 Current streak: 7 daily periods
```

Analyzing Performance:

```
User: analyze
System: [Shows analytics menu 1-7]
User: 3
System: 🏆 Longest streak: 14 periods
        Achieved by: Reading (daily)
```

---

5. Key Algorithms

5.1 Streak Calculation (Daily Habits)

Current Streak Algorithm:

1. Start from today's date
2. Check if habit was completed today
3. If yes, increment streak and check yesterday
4. Continue backwards until finding a gap
5. Return total consecutive days

Longest Streak Algorithm:

1. Convert completions to set of unique dates
2. Sort dates chronologically
3. Iterate through dates, checking for consecutive days
4. Track maximum consecutive sequence
5. Return longest sequence found

5.2 Broken Habit Detection

For Daily Habits:

- Habit is broken if: (today - last_completion) > 1 day

For Weekly Habits:

- Calculate week number for today and last completion
- Habit is broken if: (current_week - last_completion_week) > 1

---

6. Technology Stack

Component	Technology	Justification
Programming Lang	Python 3.7+	Course requirement, excellent OOP/FP support
Database	SQLite3	Built-in, reliable, professional
Testing Framework	pytest	Industry standard, powerful assertions
CLI Framework	Built-in input	Simple, no external dependencies
Date/Time Handling	datetime module	Standard library, timezone-aware

No external dependencies required for core functionality (only pytest for testing), making the application lightweight and easy to deploy.

---

7. Testing Strategy

7.1 Unit Testing Approach

Test Coverage Areas:

1. **Habit Class Tests**
    - Habit creation with valid/invalid parameters
    - Completion tracking
    - Streak calculations (current and longest)
    - Broken habit detection
    - Weekly vs daily behavior differences
2. **Database Tests**
    - CRUD operations (Create, Read, Update, Delete)
    - Completion persistence
    - Filtering by periodicity
    - Data integrity
3. **Analytics Tests**
    - All functional programming functions
    - Edge cases (empty data, single habit, etc.)
    - Calculation accuracy
    - Filter and sort operations

7.2 Test Data Strategy

- Use pytest fixtures for reusable test data
- Create temporary database for isolated testing
- Generate synthetic completion data for streak testing
- Test boundary conditions (0 completions, perfect streaks, etc.)

---

8. Design Decisions & Justifications

8.1 Why OOP for Core Classes?

Habits are natural entities with:

- State (name, periodicity, completions)
- Behavior (calculate streaks, check if broken)
- Lifecycle (created, updated, deleted)

Object-oriented design provides clear encapsulation and intuitive modeling of these real-world concepts.

8.2 Why Functional Programming for Analytics?

Analysis operations are:

- Stateless transformations of data
- Composable (build complex from simple)
- Testable (pure functions, predictable outputs)
- Parallelizable (potential future optimization)

Functional programming naturally fits these requirements and demonstrates proficiency in the paradigm.

8.3 Why CLI over GUI?

Advantages:

- Faster development (no UI framework needed)
- Platform-independent
- Scriptable and automatable
- Lightweight resource usage
- Focus on core logic rather than presentation

The CLI provides all required functionality while keeping the project scope manageable within the course timeframe.

---

9. UML Diagram

[INSERT CLASS DIAGRAM HERE showing:

- Habit class with attributes and methods
- DatabaseManager class
- HabitTrackerCLI class
- Relationships between classes
- analytics module as separate functional component]

Diagram Elements to Include:

- Class boxes with attributes (+name, +periodicity, etc.)
- Methods for each class
- Associations (CLI uses DatabaseManager, DatabaseManager manages Habits)
- Multiplicity (1 DatabaseManager manages * Habits)
- Dependency arrow from CLI to analytics module

---

10. Expected Challenges & Solutions

Challenge	Solution
Accurate streak calculation across DST changes	Use date-only comparisons, not datetime
Handling concurrent access to database	SQLite's built-in locking mechanism
Testing time-dependent logic	Parameterize dates in functions, use fixed test dates
User input validation	Try-except blocks with clear error messages
Maintaining FP purity in analytics	Never modify input collections, always return new ones

---

11. Success Criteria

The project will be considered successful when:

1. All acceptance criteria from assignment document are met
2. 5 predefined habits with 4 weeks of data are created
3. Users can create, complete, and analyze habits via CLI
4. Streak calculations are accurate for daily and weekly habits
5. Analytics functions use pure functional programming
6. All unit tests pass (target: 100% critical path coverage)
7. Code is well-documented with docstrings
8. README provides clear installation and usage instructions

Conclusion

This conception phase establishes a solid foundation for the habit tracker application. The design balances object-oriented and functional programming paradigms effectively, uses professional-grade persistence with SQLite, and provides a clear user interaction model through the CLI. The architecture is modular, testable, and extensible for future enhancements while meeting all current requirements within the project scope.