Habit Tracker Application: Development Abstract

- **Student Name**: Happiness Oribhunuebho
- **Matriculation Number**: 92133053
- **Course**: DLBDSOOFPP01 – Object Oriented and Functional Programming with Python
- **Institution**: IU International University of Applied Sciences
- **Submission Date**: 20th, Jan. 2026

1. Introduction

This abstract documents the development process of a habit tracking application created as the portfolio examination for the course Object Oriented and Functional Programming with Python. The application enables users to create, monitor, and analyze personal habits through a command-line interface, with persistent data storage implemented using SQLite. This document provides an overview of the technical implementation, development challenges encountered, and reflections on the learning outcomes achieved.

2. Technical Implementation

2.1 System Architecture

The application implements a three-layer architecture comprising presentation, business logic, and data access layers. The presentation layer consists of the HabitTrackerCLI class, which manages user interaction through a command-based interface. The business logic layer centers on the Habit class, which encapsulates habit-related data and behavior using object-oriented programming principles. The data access layer is implemented through the DatabaseManager class, which abstracts all database operations and provides a clean interface for persistence operations.

The analytics functionality was implemented as a separate module following functional programming paradigms. This module contains pure functions that transform habit data without causing side effects, utilizing higher-order functions such as map, filter, and reduce for data manipulation and analysis.

2.2 Object-Oriented Design

The Habit class serves as the core domain entity, encapsulating attributes including habit name, periodicity (daily or weekly), creation timestamp, and completion records. The class provides methods for adding completions, calculating current and longest streaks, and determining whether a habit has been broken based on missed periods. Input validation ensures that only valid periodicity values are accepted, maintaining data integrity at the object level.

The DatabaseManager class implements the persistence layer, handling all SQLite database operations. This class manages database schema creation, executes CRUD operations for habits and completions, and ensures data integrity through proper transaction management. The implementation of Python's context manager protocol enables safe resource handling and automatic connection cleanup.

2.3 Functional Programming Implementation

The analytics module demonstrates functional programming principles through the exclusive use of pure functions. Each function accepts habit data as input and returns transformed data without modifying the original collections or maintaining internal state. Complex analytical operations are constructed by composing simpler functions, demonstrating the composability principle of functional programming.

Key analytical functions include filtering habits by periodicity, calculating completion rates over specified time periods, identifying habits with low performance metrics, and generating comprehensive statistical summaries. All transformations utilize immutable data structures, with functions returning new collections rather than modifying existing ones.

2.4 Data Persistence Strategy

SQLite was selected as the persistence mechanism due to its relational structure, ACID compliance, and integration with Python's standard library. The database schema consists of two normalized tables: the habits table stores habit metadata (name, periodicity, creation date), while the completions table records individual completion events with foreign key references to their associated habits. This normalized structure prevents data duplication and enables efficient querying for time-series analysis required by streak calculations.

An index on the habit_id and completed_at columns optimizes query performance for the frequent filtering operations required by analytical functions. All datetime values are stored in ISO 8601 format to ensure consistent timezone handling and cross-platform compatibility.

3. Development Process

3.1 Implementation Approach

Development followed the three-phase structure specified in the assignment guidelines. The conception phase established the architectural design, defined the class structure, and specified the database schema. The development phase focused on implementing the core functionality, beginning with the Habit class and database layer before progressing to the command-line interface and analytics module. The finalization phase involved comprehensive testing, documentation refinement, and preparation of deployment materials.

A test-driven development approach was partially employed, with unit tests written alongside implementation code to validate functionality incrementally. This approach proved beneficial in catching edge cases early in the development cycle, particularly for the complex streak calculation algorithms.

3.2 Successful Aspects

The modular architecture facilitated efficient development by enabling work on individual components in isolation. The clear separation of concerns between presentation, business logic, and data access layers meant that each component could be implemented and tested independently before integration.

The streak calculation algorithms functioned correctly upon initial integration following careful planning during the conception phase. The decision to standardize on ISO date formats and establish clear conventions for week boundaries (treating Monday as the first day of the week) eliminated potential ambiguity in the implementation.

The functional programming implementation in the analytics module proved highly maintainable and testable. Pure functions with no side effects simplified unit testing, as each function could be tested in isolation with predictable outputs for given inputs. The use of higher-order functions resulted in concise, readable code that clearly expresses the intent of each analytical operation.

3.3 Challenges and Solutions

Several technical challenges were encountered during development. The calculation of weekly habit streaks presented complexity due to edge cases involving week boundaries and the need to determine whether completions in different calendar weeks should be considered consecutive. This was addressed by normalizing all dates to the Monday of their respective weeks and implementing consistent arithmetic for week-to-week comparisons.

The longest streak calculation algorithm required refinement to correctly handle gaps in completion data. The initial implementation incorrectly counted non-consecutive completions as part of a single streak. The solution involved sorting unique completion dates chronologically and explicitly checking for single-day or single-week differences between consecutive entries.

Testing time-dependent logic presented challenges, as habit tracking inherently involves current date comparisons. This was resolved by parameterizing all datetime dependencies in the code, allowing unit tests to inject fixed test dates rather than relying on system time. This approach ensured that tests produce consistent results regardless of when they are executed.

The command-line interface underwent multiple iterations to improve usability. Initial designs featured cryptic command structures and insufficient feedback. The final implementation incorporates clear status indicators, comprehensive help text, and intuitive command naming to enhance the user experience.

4. Testing and Quality Assurance

The application includes a comprehensive unit test suite implemented using the pytest framework. The test suite comprises 28 individual test cases covering all critical functionality including habit creation, streak calculations, database operations, and analytical functions. Tests achieve over 90 percent coverage of critical code paths, providing confidence in the correctness of core functionality.

Test fixtures were employed to generate reusable test data and manage test database lifecycle. A temporary database is created for each test session and automatically removed upon completion, ensuring test isolation and preventing side effects between test runs. Parameterized tests validate behavior across various input scenarios, including edge cases such as empty habit lists, single-element collections, and boundary conditions in date calculations.

All code includes comprehensive docstrings following Python documentation conventions. Type hints are employed on function signatures to improve code clarity and enable static type checking. The codebase adheres to PEP 8 style guidelines and maintains consistent naming conventions throughout.

5. Reflection on Learning Outcomes

5.1 Object-Oriented Programming

This project reinforced understanding of object-oriented design principles. The Habit class demonstrates proper encapsulation by maintaining private state and exposing functionality through well-defined methods. The implementation follows the Single Responsibility Principle, with each class having a clearly defined purpose within the system architecture.

The project provided practical experience in designing class interfaces that are both intuitive to use and maintainable over time. The decision to separate concerns across multiple classes rather than implementing a monolithic design improved code organization and testability.

5.2 Functional Programming

The functional programming implementation deepened understanding of pure functions, immutability, and function composition. The experience of building complex analytical operations from simple, composable functions demonstrated the power and elegance of the functional paradigm.

Working with higher-order functions such as map, filter, and reduce provided insight into declarative programming styles. The functional approach in the analytics module resulted in code that is more concise and expressive than equivalent imperative implementations would have been.

5.3 Software Engineering Practices

The project reinforced the importance of systematic design before implementation. The time invested in the conception phase, including architectural planning and database schema design, prevented costly refactoring during later development stages.

The experience highlighted the value of comprehensive testing for maintaining code quality and enabling confident refactoring. The presence of a robust test suite made it possible to optimize implementations and fix bugs without fear of introducing regressions.

Documentation proved essential for creating a maintainable codebase. The process of writing clear docstrings and usage instructions improved understanding of the code's purpose and behavior, both for potential users and for future maintenance.

6. Conclusion

The habit tracker application successfully demonstrates the application of both object-oriented and functional programming paradigms in Python. The implementation meets all specified acceptance criteria while maintaining high standards of code quality and documentation.

The development process validated the importance of upfront architectural design, iterative refinement based on testing feedback, and comprehensive documentation. The resulting application is not merely an academic exercise but a functional tool that effectively tracks habits and provides meaningful analytical insights.

The integration of object-oriented design for entity modeling and functional programming for data transformation proved to be an effective combination, leveraging the strengths of each paradigm appropriately within different layers of the application architecture.

References

GitHub Repository:

https://github.com/herthiqal/Oribhunuebho_Happiness_92133053_OOFPP_Habits_Submission_Final