**Summary Essay**

Module Code: FC723

Class/Group: Foundation Certificate for Science and Engineering | Group B

Module Title: Programming Theory

Assessment Type: Project

Module Tutor Name: Callum Birkett

Student ID Number: P470158

Date of Submission: 2025.4.4

GitHub link: <https://github.com/Ddddd917/apache-airlines-seat-booking>

GitHub link: 👉 <https://github.com/Ddddd917/apache-airlines-seat-booking>

I confirm that this assignment is my own work.

Where I/we have referred to academic sources, I have provided in-text citations and included the sources in the final reference list.

# Introduction

The Apache Airlines Seat Booking System is a command-line based Python application developed as part of the FC723 Software Development module. The aim of the project was to design, implement, and refine a modular software system capable of handling typical airline seat reservation processes.

**A diagram of a person with text

Description automatically generated**The system was developed to simulate a realistic airline seat booking experience using a text-based interface. It allows users to interact with a Burak757 aircraft layout, providing them with options to check seat availability, book or cancel seats, and view overall booking statistics. The user interface is menu-driven and supports interactive input throughout the program’s lifecycle.

From the beginning, the project followed a structured approach to software development. It incorporated key programming concepts such as **requirement analysis**, **modular design**, **data validation**, **persistent data storage using SQLite**, and **version control via GitHub**. Over four versions of progressive development, the system evolved from a basic reservation tool to a feature-rich application demonstrating object-oriented design and database integration.

This summary essay documents the development journey, highlights the design decisions, and reflects on the testing and refinement phases that led to a complete, functional, and maintainable seat booking system.

**Use Case Diagram**

# Requirement Analysis

The development of the Apache Airlines Seat Booking System was guided by both functional and non-functional requirements, identified at the start of the project and expanded upon in later iterations. Because of the size of the picture, it cannot be clearly displayed in the document, please check the Diagram folder

**Functional Requirements**

The system was initially required to support five core functionalities, forming the basis of **Version 1**:

• **Check seat availability**: Users can input a seat ID (e.g., “12A”) to check whether the seat is free or reserved.

• **Book a seat**: Users can reserve a specific seat. The system confirms the booking if the seat is available.

• **Cancel a booking**: A previously reserved seat can be released and marked as available again.

• **Display seat layout**: The full seating map of the Burak757 aircraft is displayed, showing the status of each seat.

• **Exit program**: Users can safely terminate the application from the menu.

A diagram of a flowchart

Description automatically generated

***Activity Diagram of Version 1***

In **Version 2**, three user-centered enhancements were added:

• **Group booking**: Users can reserve up to three adjacent seats in one transaction, streamlining the process for families or groups.

• **Seat preference search**: Based on user input (window, aisle, or middle), the system recommends suitable available seats, prioritizing front-row options.

• **Booking summary**: When viewing the seat map, users are also shown booking statistics (e.g., total seats, booked seats, available seats).

A diagram of a company

Description automatically generated

***Activity Diagram of Version 2***

**Version 3** introduced data persistence and security enhancements:

• **Database integration**: All bookings are stored in a local SQLite3 database, along with passenger name and passport number.

• **Unique reference code**: Each booking generates a unique 8-character alphanumeric code, used for future operations like cancellation.

• **Identity-based search**: If a user forgets their reference code, they can retrieve it by entering their full name and passport number.

A diagram of a company

Description automatically generated

***Activity Diagram of Version 3***

In **Version 4**, the focus shifted to improving code structure and maintainability:

• **Object-Oriented Programming (OOP)**: The system was fully refactored using class-based architecture. Each module was encapsulated into clearly defined classes such as BookingManager, SeatMap, and DatabaseManager.

• **Improved modularity and separation of concerns**: Functions were grouped into logical classes with single responsibilities, enhancing code readability and reusability.

• **Main program class**: A BookingSystem class was introduced to manage all interactions and coordinate other modules cleanly.

• **Validation enhancements**: Input validation and data verification were fully integrated into the object-oriented structure for consistency across modules.

**Non-Functional Requirements**

To ensure the system’s robustness and user-friendliness, several non-functional requirements were also defined:

• **Usability**: A clear, menu-based interface with intuitive prompts.

• **Performance**: All operations should execute within one second under normal conditions.

• **Validation**: User input (e.g., seat ID, passport number, reference code) must be strictly validated to prevent crashes or inconsistent data.

# Development Process & Versioning

The development of the Apache Airlines Seat Booking System followed an iterative, Agile-inspired approach, divided into four clearly defined versions. Each version introduced meaningful improvements in functionality, usability, and structure.

A screenshot of a computer

Description automatically generated

***GitHub Commit History***

**This screenshot shows the commit messages and version tags (v1.0 to v4.0) created using GitHub Desktop to track project progress.**

**Version 1** focused on implementing the essential features: checking seat availability, booking and cancelling a seat, displaying the seat layout, and exiting the program. The codebase in this stage was organized into a few functional modules (main.py, seatmap.py, and booking.py) using procedural programming techniques.

A screenshot of a computer

Description automatically generated

***Folder Structure in Version 1***

**Early code organization with only basic Python files in a flat structure.**

**Version 2** introduced enhanced functionality aimed at improving the user experience. These included group booking for adjacent seats, seat preference search (window/aisle/middle), and a booking summary displaying total, booked, and available seats. Additional Python modules such as group.py, preference.py, and summary.py were introduced, increasing the modularity of the system.

A screenshot of a computer

Description automatically generated

***Folder Structure in Version 2***

**This version introduced additional feature-specific modules, enhancing code separation and feature traceability.**

**Version 3** marked a major turning point by integrating **SQLite3** for data persistence. A new database.py module was added to handle storage of passenger information and booking references. A unique 8-character alphanumeric reference code was introduced for every booking, and identity-based search functionality was implemented to retrieve bookings using a user’s name and passport number. Input validation functions were centralized in a new module validation.py, and constants were extracted into a constants.py file for easier maintenance.

A screenshot of a computer

Description automatically generated

***Folder Structure in Version 3***

**More advanced modular layout introduced with database integration and validation modules.**

**Version 4** focused on software quality and maintainability. The entire system was refactored using **Object-Oriented Programming (OOP)** principles. All key logic was encapsulated into classes such as BookingManager, SeatMap, and DatabaseManager, while the BookingSystem class was introduced to coordinate all program logic. This restructuring improved code readability, reusability, and future scalability.

A screenshot of a computer

Description automatically generated

***Folder Structure in Version 4 (OOP)***

**Final refactored version showing class-based module separation for maintainability.**

Version control was managed using **GitHub Desktop**, with meaningful commit messages and version tags (v1.0, v2.0, etc.) applied at each phase. Each version also included a changelog text file (e.g., version\_3\_description.txt) documenting new features and improvements.

In addition to functional development, I paid special attention to project documentation and version tracking. For every major version, I created a dedicated version\_x\_description.txt file outlining the changes, new features, and design decisions. These changelogs provide a clear and traceable development history, which is especially useful for long-term maintenance or code reviews.

Furthermore, I continuously updated the README.md file to reflect the current system capabilities, version progression, and technical structure. This not only met the module requirements but also allowed me to treat the repository as a **personal programming portfolio**, showcasing my ability to manage, document, and deliver a complete software project. I intend to include this repository in future academic or job applications as evidence of my development skills.

# Implementation & System Design

To enhance maintainability and align with professional development practices, the system was fully refactored using Object-Oriented Programming (OOP) in Version 4. Each major functionality was encapsulated in dedicated classes with clearly defined responsibilities, enabling better structure, readability, and future scalability.

**Class Overview and Responsibilities**

The final system design consists of seven main classes:

**BookingSystem**

• **Role**: Acts as the main controller of the entire program.

• **Attributes**:

• seat\_map: SeatMap — instance of the seat layout.

• manager: BookingManager — handles all booking logic.

• **Methods**:

• run ( ) — main loop to start the system.

• show\_menu ( ) — displays available operations.

***Associates with:*** *BookingManager, SeatMap, SearchManager, SeatPreference, DatabaseManager, Validator*

**BookingManager**

• **Role**: Handles all booking-related operations.

• **Attributes**:

• db: DatabaseManager — manages communication with the database.

• **Methods**:

• book\_seat ( )

• cancel\_by\_reference ( )

• group\_booking ( )

• generate\_reference\_code ( ) — generates a unique 8-character code.

*Associates with: DatabaseManager*

*Depends on: Validator for input checks*

**SeatMap**

• **Role**: Manages the internal data structure for seat status and layout rendering.

• **Attributes**:

• seats: dict — key-value map of seat IDs and their statuses.

• **Methods**:

• initialize\_seats ( )

• display ( )

• get\_summary ( ) — returns total, booked, and available seats.

*Used by: BookingSystem and SeatPreference*

**DatabaseManager**

• **Role**: Handles all SQLite database operations.

• **Methods**:

• insert\_booking(name, passport, seat\_id, reference)

• get\_booking\_by\_reference(reference)

• delete\_booking(reference)

*Used by: BookingManager, SearchManager*

**Validator (Static Class)**

• **Role**: Provides input validation functions.

• **Methods**:

• is\_valid\_seat\_id ( )

• is\_valid\_passport\_number ( )

• is\_valid\_reference\_code ( )

*Called by: BookingSystem, BookingManager*

**SearchManager**

• **Role**: Allows searching bookings by identity information.

• **Methods**:

• search\_by\_identity(name, passport)

*Uses: DatabaseManager*

**SeatPreference**

• **Role**: Recommends available seats based on user preference (window/aisle/middle).

• **Methods**:

• recommend\_seats(pref: str)

*Accesses: SeatMap data indirectly*

**Class Relationships**

The class diagram below summarizes the system’s internal structure and the interactions between modules:

A diagram of a server

Description automatically generated with medium confidence

***Class Diagram :Object-Oriented Design of Version 4***

**Each class is shown with its attributes and methods. Solid arrows represent associations, while dashed arrows represent dependencies.**

• BookingSystem is the central orchestrator, associated with all major classes.

• BookingManager has a strong association with DatabaseManager and relies on Validator.

• SeatMap is a standalone class accessed by both BookingSystem and SeatPreference.

• SearchManager and SeatPreference serve as utility modules extending functionality.

This object-oriented structure significantly improves code readability, supports modular growth, and mirrors real-world responsibilities, making the system easier to test and maintain.

**Database Design & Integration**

To achieve data persistence, the system uses SQLite3, a lightweight and file-based relational database. A single database file named flight757\_booking.db is created and managed through the DatabaseManager class, which encapsulates all database interactions.

**The database contains one main table called passengers, which stores all relevant booking information, including:**

• reference\_code (TEXT): A unique 8-character alphanumeric identifier for the booking

• name (TEXT): The full name of the passenger

• passport (TEXT): A unique passport number used as a secondary identifier

• seat\_id (TEXT): The reserved seat (e.g., “12B”)

• timestamp (TEXT): The exact time the booking was made, stored in ISO format

Each time a booking is confirmed through the BookingManager, a new record is inserted into the passengers table. When a booking is cancelled, the corresponding row is deleted. This ensures that the in-memory seat status and the persistent data storage remain synchronized.

The inclusion of a timestamp field provides an additional layer of traceability, allowing for future enhancements such as booking history, audit logs, or usage analysis.

**The database design supports additional features such as:**

• Identity-based search, using a combination of name and passport

• Validation of reference code uniqueness, by checking for existing records before accepting a new booking

• Modular access, with all interactions performed via structured queries inside DatabaseManager, ensuring data integrity and security

A screenshot of a computer

Description automatically generated

***Passengers Table in SQLite Database***

# Testing and Evaluation

Thorough testing was conducted throughout the development process to ensure the Apache Airlines Seat Booking System was reliable, user-friendly, and functionally correct. The system was tested manually using a range of realistic scenarios via terminal-based input.

**Manual Testing Strategy**

Each major function was tested using:

• **Normal cases**: e.g., booking a valid seat like 12A

• **Edge cases**: e.g., first row 1A, last row 80F, or storage-only rows (77D)

• **Invalid inputs**: e.g., invalid seat ID (99Z), too-short passport number (A1), malformed reference codes

The test cases were executed manually by navigating the system menu and entering input as prompted. After each operation, the system’s output was observed to verify correctness, and the SQLite database was checked to confirm data integrity.

A screenshot of a computer program

Description automatically generated

***Terminal flow of Check Seat Availability***

A screenshot showing how the system responds when a user checks a seat, system print seat statues message, and then back to main menu.

**Test Case Table**

To ensure all key features were covered, a set of test cases was designed to cover every menu function. An example testing table is shown below:

A screenshot of a computer screen

Description automatically generated

***Manual Test Case Table***

**Observations**

• The system **prevented invalid operations** such as booking storage seats (e.g., 77F) or using an incorrect seat ID format.

• Booking a seat updated both the **terminal display** and the **passengers table in SQLite** accurately.

• Cancelling a booking correctly deleted the row in the database and freed the seat in memory.

• The seat recommendation system provided front-row, preference-matching results as expected.

• Identity-based search successfully retrieved booking information when the reference code was forgotten.

**Limitations**

• No automated unit tests (e.g., via unittest or pytest) were implemented. All testing was interactive and manual.

• Error messages were printed in the terminal but not logged persistently.

• Some input validation logic could be further extended (e.g., stricter name formatting rules).

Despite these limitations, the system demonstrated strong stability and correctness across repeated usage.

# Conclusion and Reflection

The development of the Apache Airlines Seat Booking System has been a valuable and insightful learning experience. It allowed me to apply key principles of software engineering—ranging from requirement analysis and modular programming to data persistence and object-oriented design—within the context of a real-world-inspired application.

From the outset, I focused on building a system that was not only functional but also maintainable and extensible. The early procedural implementation in Version 1 helped me understand the core logic of menu-driven user interactions. As the project evolved through Versions 2 and 3, I introduced more sophisticated features such as group booking, seat preference search, and a persistent SQLite database. By Version 4, I refactored the entire codebase using Object-Oriented Programming, introducing well-structured classes like BookingSystem, BookingManager, and DatabaseManager. This transformation greatly improved the clarity, modularity, and scalability of the system.

One of the biggest technical challenges I faced was ensuring the uniqueness and consistency of booking reference codes. I had to design a mechanism that generated random codes and verified them against the database before confirming a booking. Another challenge was managing seat state between memory and database synchronization, especially during group bookings and cancellations. Careful validation and testing helped address these issues.

Throughout the project, I also made a deliberate effort to treat the codebase as a professional portfolio. I documented each version with dedicated changelog files (e.g., version\_2\_description.txt) and maintained a detailed README.md to explain functionality, structure, and usage. Using GitHub Desktop for version control allowed me to organize my progress effectively, track each improvement, and ensure that each commit reflected meaningful development steps.

**Future Improvements**

While the current system meets all the functional requirements, there is room for enhancement:

• Transition to a graphical user interface (GUI) using tools like Tkinter or PyQt to improve user experience.

• Implement an admin dashboard with analytics or CSV export capabilities.

• Introduce email confirmations using SMTP for booking success notifications.

• Add automated testing frameworks (e.g., unittest, pytest) to streamline the verification process.

• Include login or authentication features to differentiate between users or roles.

**Final Thoughts**

This project deepened my understanding of Python programming, database integration, and system design. It gave me a sense of how real-world applications are developed iteratively and improved over time. More importantly, it showed me the importance of **planning, testing, and documenting**—skills that will benefit me in future academic and professional settings. I am proud to include this work in my digital portfolio as a demonstration of both technical ability and personal growth.