**Reflective Essay**

**Module Code:** FC723

**Class/Group:** Group 1

**Module Title:** Programming Theory

**Assessment Title:** Project - Application

**Tutor Name:** Sophie Norman

**Student GUID Number:** P460084

**Date of Submission:** 6 July 2025

Developing the Apache Airlines seat-booking system on my own was both a challenging and rewarding experience, allowing me to apply and deepen a wide range of software engineering skills. Throughout this project, I was responsible for every aspect—from requirements analysis and system design to programming, testing, debugging, and documentation. Managing the entire workflow alone required not only technical knowledge but also a disciplined and structured approach to project management.

To stay organized and productive, I adopted an Agile-inspired methodology, dividing the project into weekly sprints. At the start of each sprint, I set clear objectives, such as building the graphical user interface (GUI), integrating the database, or refining error handling. I used Trello to manage tasks, breaking down requirements into user stories like “As a passenger, I want to book a seat and receive a unique reference.” Each day, I kept brief self-reflection notes to track progress and identify challenges. This habit helped me maintain focus, adapt quickly when issues arose, and continuously improve my workflow.

Version control was handled using GitHub, with a branching strategy that separated features and bug fixes into their own branches. Even as a solo developer, I found value in opening pull requests and reviewing my own code before merging. This extra step enabled me to catch potential issues such as unclear variable names, duplicated code, and logic errors. Studies confirm that regular code review, even if self-conducted, reduces bugs and improves software quality (Bacchelli & Bird, 2013). Testing also evolved over the course of the project. I started with manual testing for each function but soon adopted the pytest framework for automated unit testing. Automated tests improved reliability, made refactoring safer, and ensured that new features did not break existing ones.

Exception handling became one of the core technical challenges. In early versions, the application crashed if users entered invalid input or tried to book an already-occupied seat. I addressed this by implementing try-except blocks around all user input and database operations. Now, if a seat is already booked or invalid data is entered, the system returns a clear error message rather than failing silently or crashing. I also handled exceptions like database connection errors and attempts to cancel nonexistent bookings, which made the application much more stable and user-friendly.

A key functional requirement was the automatic generation of a unique eight-character alphanumeric booking reference for each reservation. To accomplish this, I wrote a function that queries all existing references from the database and then generates random codes until a unique one is found. With 36^8 possible codes, the likelihood of a collision is extremely small, but my approach guarantees uniqueness. For this assignment, I used Python’s random module, but for real-world deployment, a cryptographically secure method would be more appropriate.

Persistent storage was achieved with SQLite, a lightweight and widely-used database. I encapsulated all database logic in a DBManager class, which managed connecting to the database, creating tables, and handling booking and cancellation records. Using context managers for transactions ensured that any failed operation would roll back automatically, protecting data integrity. Although SQLite is ideal for a local, single-user application, I designed the code to allow easy migration to more robust systems like PostgreSQL if needed in the future.

While I worked alone, I still experienced the classic issue of merge conflicts when working on different branches at once. This taught me to make frequent, small commits and regularly pull from the main branch to keep everything synchronized. Tools like git diff and git log helped me trace changes, resolve conflicts, and maintain a clear project history.

Design documentation was a critical part of my process. I developed several UML diagrams to clarify the architecture before writing code. The use-case diagram mapped the main actors—Passenger and Booking Agent—and their interactions with the system’s features, such as seat booking and reference generation. The activity diagram modeled the flow of actions from the menu to booking and cancellation processes, including error checks and database updates. The class diagram illustrated the main components—BookingSystem, SeatMap, DBManager, and Booking—and their relationships. Keeping these diagrams updated as the project evolved made debugging and adding new features more manageable.

Usability and interface design were also important considerations. I aimed for a GUI that was simple, intuitive, and provided clear feedback. Error messages were designed to be user-friendly, helping users recover quickly from mistakes. I conducted informal usability testing with classmates, whose feedback led me to improve navigation flow and action visibility. These iterations made the system more accessible and enjoyable to use.

Security and data protection, while not a primary requirement, were still considered. I used parameterized SQL queries to prevent injection attacks and ensured passenger data was stored and retrieved safely. For a real-world system, I would add encryption for sensitive data and require user authentication for booking management features.

Reflecting on the development process, I realized that discipline and continuous improvement were key to success. Keeping daily notes helped me remain focused and respond effectively to problems as they appeared. Automated testing and regular code review increased my confidence in the application’s reliability. Regularly updating UML diagrams maintained clarity in the system’s structure and made future enhancements easier.

Looking ahead, there are several ways I could further enhance the system. Implementing continuous integration with GitHub Actions would ensure that every code change triggers automated testing and style checks. Adding GUI automation with tools like Playwright could simulate real user actions and detect interface bugs. Creating a RESTful API would allow web or mobile clients to interact with the booking system remotely. Accessibility improvements—like keyboard navigation and screen reader support—would make the application usable for all. Finally, introducing database migration tools and automated encrypted backups would boost robustness and support safe upgrades.

In conclusion, developing the Apache Airlines seat-booking application by myself required me to practice a wide range of software engineering skills. Through structured, iterative development, careful version control, automated testing, and thoughtful design documentation, I delivered a reliable and maintainable system. This project greatly improved both my technical expertise and my ability to work independently. The lessons I learned will benefit future academic and professional work, especially in planning, organizing, and reflecting on complex software projects.

**References**

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. *Proceedings of the 2013 International Conference on Software Engineering*, 712-721.

Kniberg, H., & Skarin, M. (2010). Kanban and Scrum: Making the Most of Both. C4Media.

Nielsen, J. (2020). 10 usability heuristics for user interface design. *Nielsen Norman Group*. https://www.nngroup.com/articles/ten-usability-heuristics/