Figure 1. Use Case Diagram for Apache Airlines Seat Booking System

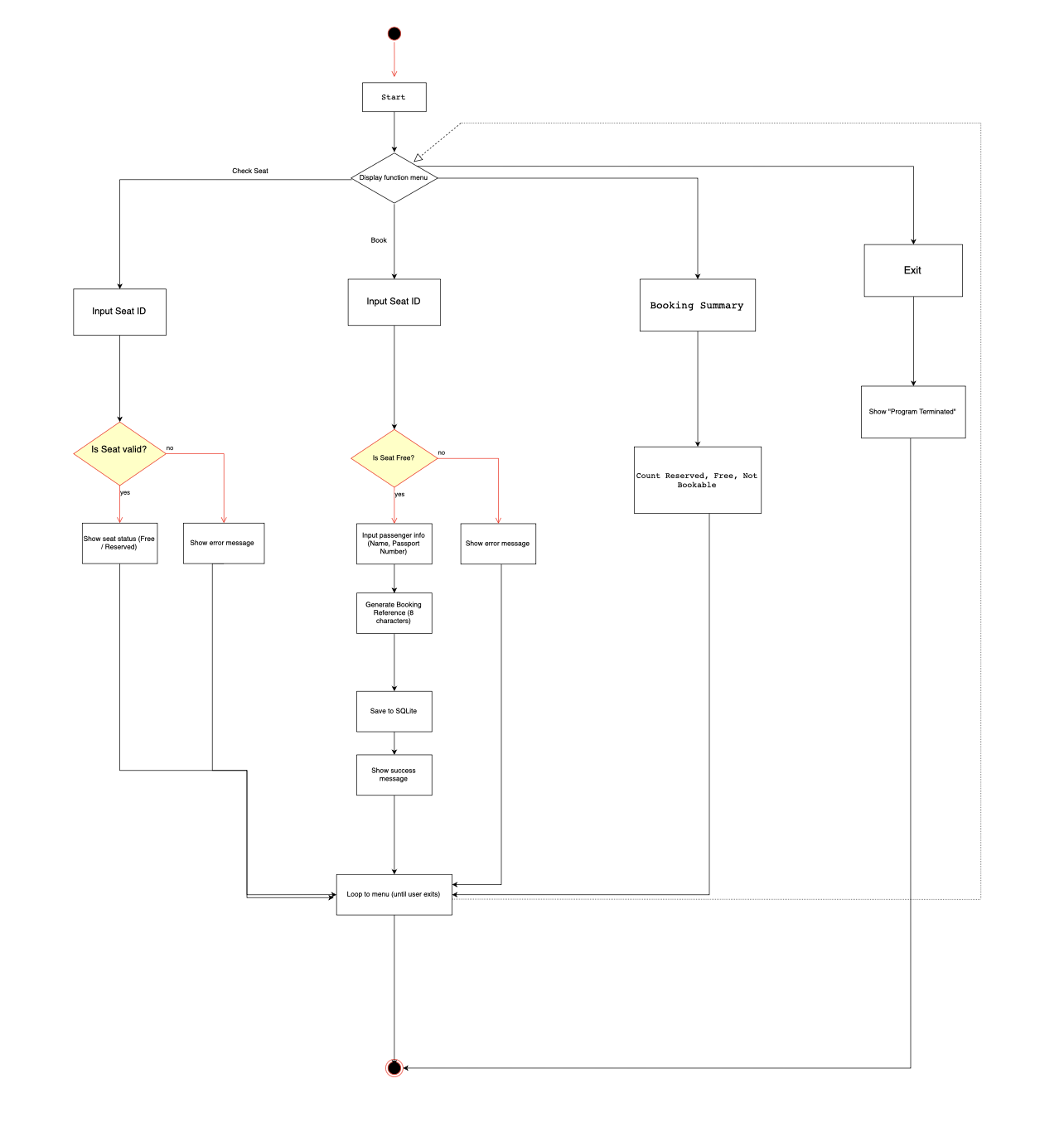


Figure 2. Activity Diagram for User Interaction Flow

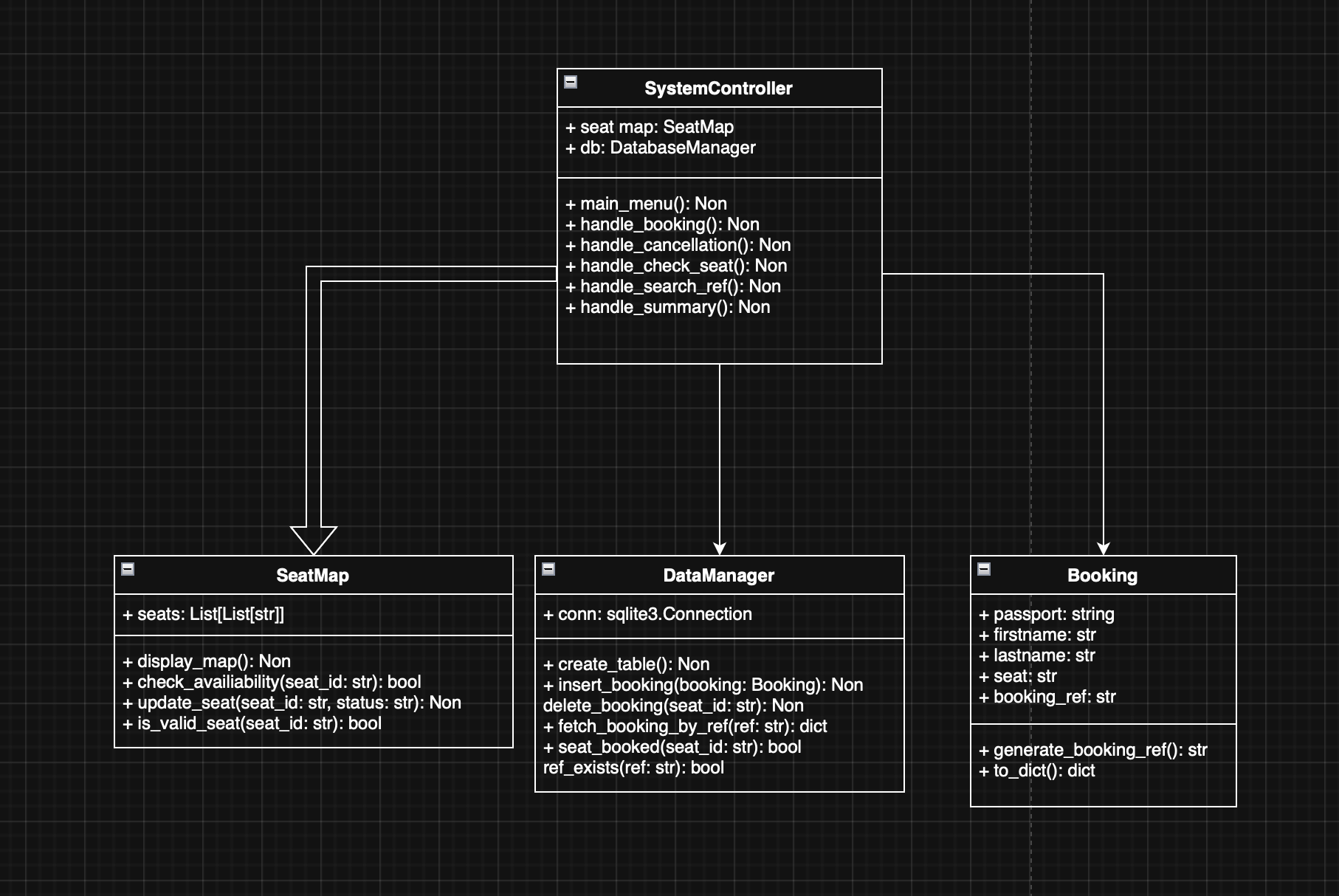


Figure 3. Class Diagram Representing the Internal Structure

The Apache Airlines Seat Booking System is a command-line Python application that allows users to check seat availability, book and cancel seats, search for bookings using reference codes, view booking summaries, and display the seat map. A straightforward menu-driven interface is used for user interaction. To cancel a reservation, for example, a user must choose the "Cancel a Booking" option, input a seat ID, and the system will update the database and seat status if the seat was previously reserved.

To help design and communicate how the system works, I created three diagrams: the Use Case Diagram (Figure 1), the Activity Diagram (Figure 2), and the Class Diagram (Figure 3). "Book a Seat" and "Search by Booking Reference" are two of the six main features that users can access, according to the Use Case Diagram (Figure 1). It made sure I was taking care of all the important user needs. The Activity Diagram (Figure 2) explains how these features work in the background. For instance, when you cancel, the system updates the display, deletes booking data from SQLite, and checks the status of your seat before bringing you back to the menu. This enhanced validation and made logic flow easier to see. The Class Diagram (Figure 3) shows internal organization: the SystemController handles menu logic, SeatMap tracks seat layout, Booking stores user details, and DataManager manages database functions. These diagrams were crucial in structuring and debugging the system. For instance, the Use Case Diagram’s six functions directly guided the features I implemented in my menu system, such as the ability to check seat availability and search bookings. The Activity Diagram helped me confirm that each process, like booking and cancelling, returned the user to the correct state and included all necessary validation steps. More significantly, the Class Diagram served as a model for my actual Python classes: Booking encapsulated user input and the creation of booking references, SeatMap managed seat layout stored as a 2D array, SystemController controlled the main loop and menu options, and DataManager managed all SQLite operations with separation of concerns. Debugging was made simpler by this obvious alignment, which also prevented me from combining database logic and UI code. By reminding me to keep logic and data separate, they also assisted me in refactoring my code. For instance, the Class Diagram helped me avoid combining database commands with user interface code by precisely matching the structure of my code modules.

Using brief two- to three-day sprints, I initially developed a development plan based on the Agile methodology. Specific objectives, such as creating the seat matrix, integrating essential features, or connecting the database, were the focus of each sprint. In every sprint, I adhered to the following mini-cycle: plan, develop, test, debug, and commit. I was able to stay on course and address problems early thanks to this iterative process. For instance, because SQLite integration and booking reference logic were more complicated, I planned them for later sprints. Although this process helped me organise my time, I will do a better job of using task boards like Trello and backlogs to track the difference between my estimated and actual task durations in the future.

One major challenge was preventing system crashes when users entered invalid seat IDs like "Z9". This caused index errors. At first, I didn’t understand why it crashed, so I printed the error traceback and debugged step-by-step. I eventually wrote an is\_valid\_seat() function to check input format and index range before proceeding with actions. This fix prevented crashes and helped provide meaningful error messages. In future iterations, I would allow users to pick from valid seat options to prevent such mistakes altogether. Another bug I encountered involved booking reference duplication. I created 8-character codes using Python's random and string modules, but when I tested them, I discovered infrequent duplicates when making fast seat reservations. I changed the reference generator to look for existing codes in the database and regenerate if a match was found in order to address this. I discovered that UUIDs or timestamp-based references are superior for high-concurrency environments, even though this works for small-scale systems.

During final testing, I encountered additional edge cases where users entered valid-looking but structurally incorrect seat IDs, such as “A10” or “Z9”. Although these inputs passed basic format checks, they either referred to non-existent columns or invalid rows, which previously caused the program to crash. These were not part of the original development plan but emerged through hands-on usage and stress testing. To address this, I implemented a more robust is\_valid\_seat\_format() function that uses regular expressions to accept multi-digit seat IDs while still enforcing a strict pattern. I also added a secondary layer of validation that checks whether the specified row and column actually exist in the seat\_matrix. This prevented inputs like “Z9” or “A100” from causing index errors. Additionally, I enhanced the seat layout display by adding column headers (e.g., “1 2 3 4 5”) to improve user navigation. These unplanned yet impactful changes reflect how Agile testing cycles and real-time feedback can lead to immediate improvements in usability, stability, and system reliability.

In early versions, I inserted user input directly into SQL queries like SELECT \* FROM bookings WHERE reference = '{ref}'. This created a security vulnerability. After researching SQL injection attacks, I refactored the code using parameterized queries such as cursor.execute("SELECT \* FROM bookings WHERE reference = ?", (ref,)), which protected the system and improved code clarity. Each module was tested immediately after development. For the seat booking feature, I tested successful bookings by reserving free seats like “A1”, and confirmed that the seat status changed to “R” and the booking reference was generated correctly. I then used the “Display Seat Map” function to verify the visual update. For edge cases, I entered already-reserved seats, like trying to book “A1” twice, which correctly returned an error message saying the seat was not available. I also tested the cancellation feature by freeing both booked and non-booked seats. When cancelling a seat like “A1”, the system successfully changed the status back to “F” and removed the associated row in the SQLite database. When cancelling an already free seat or entering an invalid code like “Z9”, the system returned a validation error without crashing. For the booking reference search, I tested it by entering correct codes, lowercase codes, and invalid codes. At first, lowercase inputs like “abc123xy” failed because references were stored in uppercase. I fixed this by converting all search inputs to uppercase. After the fix, both “ABC123XY” and “abc123xy” returned the correct passenger info. I also confirmed that entering a non-existent code gave a clear “Not Found” message. For database testing, I opened booking.db using DB Browser to confirm that booking information (passport number, name, seat ID, booking ref) was correctly saved and deleted. I purposely entered incomplete data in some cases (e.g., missing passport) to make sure the system would reject it, which it did. I also checked that after cancellation, the relevant booking row was removed from the database. For the Booking Summary feature, I tested after various bookings and cancellations to verify that reserved, free, and blocked seats were correctly counted in real-time. I cross-checked these values with the current seat matrix to ensure accuracy. All test cases were run at least twice to confirm stability. In future, I plan to adopt unit testing and document tests more systematically.

A Booking Summary feature that provides statistics such as Total Seats: 36, Reserved Seats: 7, Free Seats: 25, and Non-bookable Seats: 4 (X/S) was added to enhance analysis and feedback. This feature makes it possible to quickly determine availability. This real-time feedback, while not graphical, aids administrators and users in tracking usage. To further enhance visualisation, I would think about including a heatmap or basic bar chart in a later version.

Through this project, I improved my ability to manage full-stack systems, debug critical errors, and apply database principles. At the beginning, I lacked understanding of secure database handling, but now I can independently implement and protect CRUD operations using SQLite. My confidence in code testing also increased, as I learned to trace and resolve bugs methodically. I also learned to rely on design diagrams to guide implementation, especially when mapping system logic to functions. This experience has not only strengthened my Python skills but also helped me understand software engineering principles in practice. Moving forward, I plan to apply MVC patterns earlier in design, use Git commit message standards, integrate CI/CD pipelines, shift from manual to automated testing using unittest or pytest, and explore ORM tools like SQLAlchemy for database abstraction. In future modules or internships, I believe these skills will help me work more confidently in teams, explain my design decisions clearly, and build scalable applications.

I was able to advance from writing functions to creating and overseeing a reliable system thanks to this project. More significantly, it caused me to consider the shortcomings of my initial choices. For instance, I didn't use any formal design patterns at first, which resulted in code that was more difficult to maintain and extend. I was better able to comprehend the importance of separation of concerns after reworking the system to isolate database logic from menu interaction. It also helped me realise that well-designed software architecture involves code that is easily scalable, reusable, and testable, not just code that works. For instance, selecting SQLite over JSON made it easier for me to see the benefits of relational queries and data persistence, particularly when I eventually had to retrieve data using reference codes. Trial and error were the only way to gain these insights. Technical difficulties are now viewed by me as teaching opportunities that enhance my design and thought processes rather than as barriers. Any future software engineering position will require this type of critical thinking, which includes weighing trade-offs, identifying errors early, and improving architecture. I learnt that scalable, maintainable code is the result of careful planning, modular thinking, and proactive debugging. I now feel more comfortable taking on challenging software engineering tasks, both professionally and academically.