CSYE 7374- DESIGN PATTERNS

GROUP 13

TRAVEL RESERVATION SYSTEM

NISHI PANCHOLI
ANDREA SEQUEIRA
ATHARVA KAMBLE
AZIZ VOHRA
NEHA BALAJI
SATYAJIT DAS

IMPLEMENTED DESIGN PATTERNS

Why Design Patterns are important?

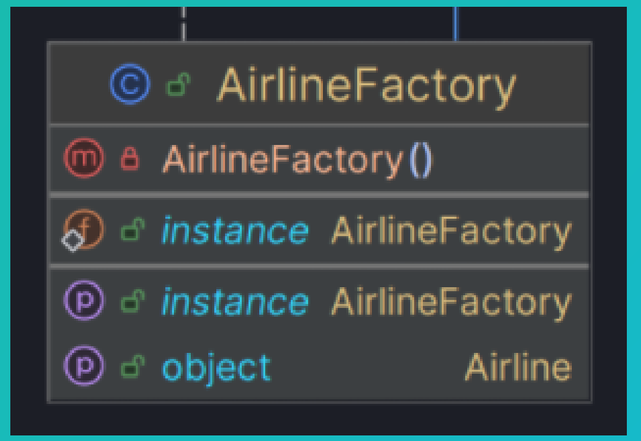
Design patterns are crucial as they provide reusable solutions to common design problems, improve code readability and maintainability, promote scalability, and ensure adherence to industry best practices, enabling efficient and robust software development.

Implemented Design Patterns:

- Singleton
- Factory
- Builder
- Facade
- Strategy
- Decorator
- Bridge
- Prototype
- Command
- Observer
- Adapter
- State

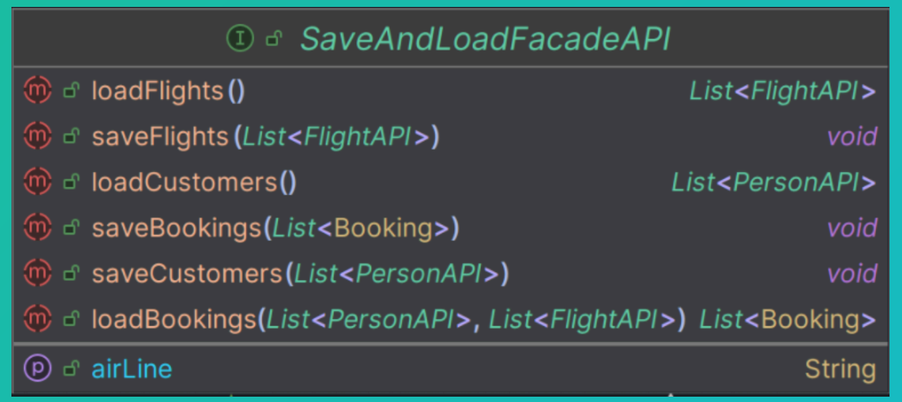
SINGLETON + FACTORY DESIGN PATTERN

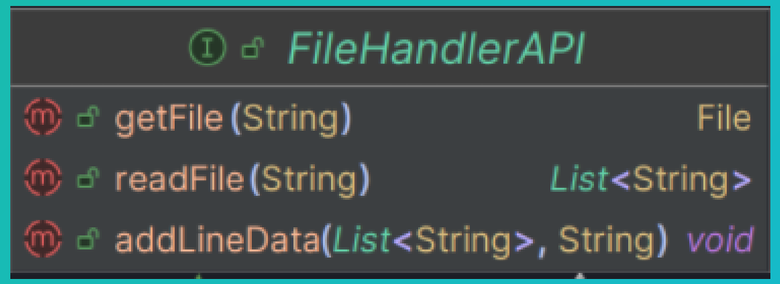
- The *main* method invokes the *demo* method to demonstrate functionality.
- Inside the *demo* method, an *Airline* object is instantiated using the *AirlineFactory*.
- The *AirlineFactory* implements the Eager Singleton pattern, ensuring only one instance is responsible for producing *Airline* objects.



FACADE DESIGN PATTERN

- The ***SaveAndLoadFacadeAPI*** abstracts the complexities of saving and loading data, simplifying program startup and shutdown processes for the user.
- It supports multiple data-saving methods, including an implementation for CSV file handling.
- An instance of the ***SaveAndLoadFacadeAPI*** is seamlessly integrated into the ***Airline*** object for efficient data management.



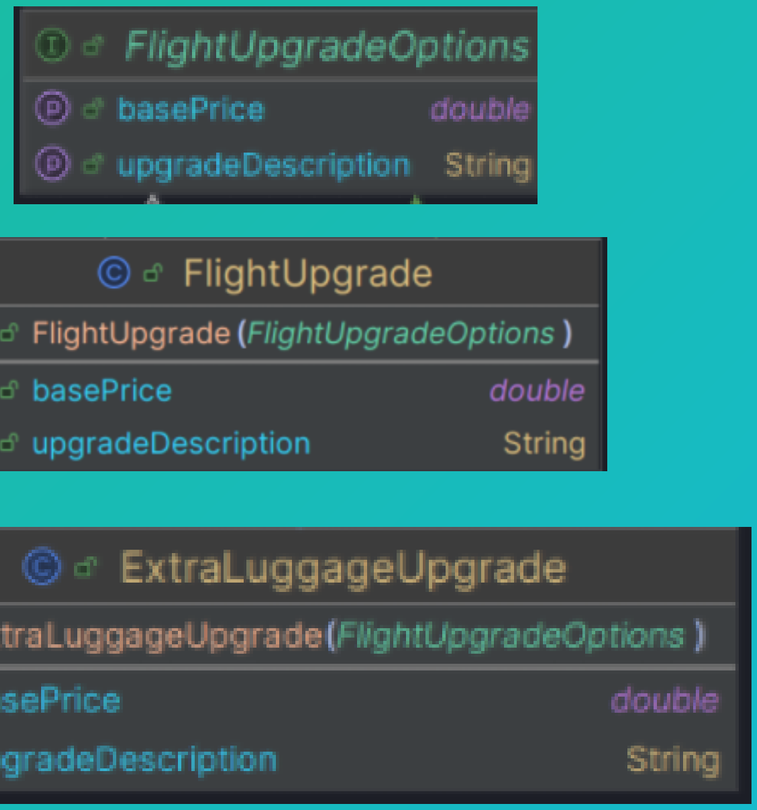


ADAPTER DESIGN PATTERN

- **Currency Flexibility:** The client can view flight prices in multiple currencies (e.g., INR, Canadian Dollars, US Dollars), providing a localized user experience.
- **CurrencyAdapter Functionality:** The *CurrencyAdapter* leverages an instance of the *Currency* interface to adapt and convert flight prices to the desired currency seamlessly.
- **Legacy API Integration:** The *CurrencyAdapter* encapsulates the legacy API, which provides prices in US Dollars, ensuring compatibility without exposing the legacy implementation details to the client.

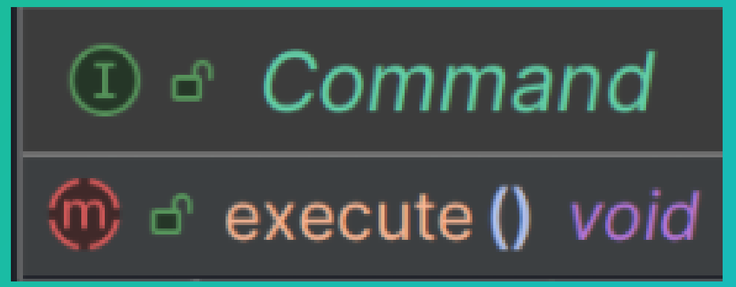
BRIDGE DESIGN PATTERN

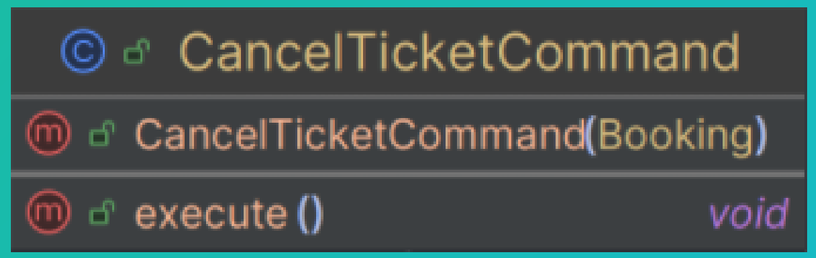
- The Bridge pattern decouples abstraction (*TicketFeature*) from its implementation (*SeatChangeFeature* and *MealPreferenceFeature*) for independent evolution.
- It enhances flexibility by allowing changes or extensions to abstraction and implementation hierarchies without mutual interference.
- Demonstrated in the *Demo* class, this pattern enables easy functionality extension without altering existing code.

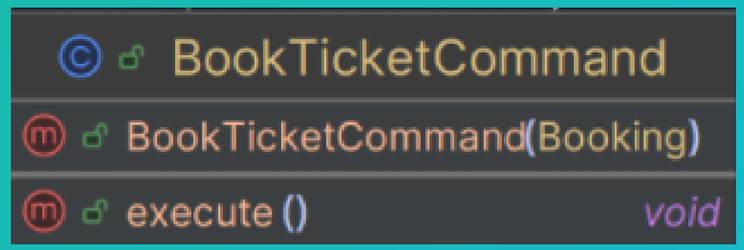


COMMAND DESIGN PATTERN

- **Command Interface: The *Command* interface defines the *execute*() method, which is implemented by concrete command classes for encapsulating operations.**
- **BookTicketCommand: Calls the *bookTicket*() method on the *Booking* object.**
- **CancelTicketCommand: Calls the *cancelTicket*() method on the *Booking* object.**
- **Invoker Role: The *TicketInvoker* stores and triggers commands, allowing the client to set instances of *BookTicketCommand* and *CancelTicketCommand* dynamically.**
- **Benefits: The Command pattern decouples the object invoking the operation (*Invoker*) from the one performing it (*Receiver*), enabling flexibility, undo/redo actions, and easily extensible functionality.**





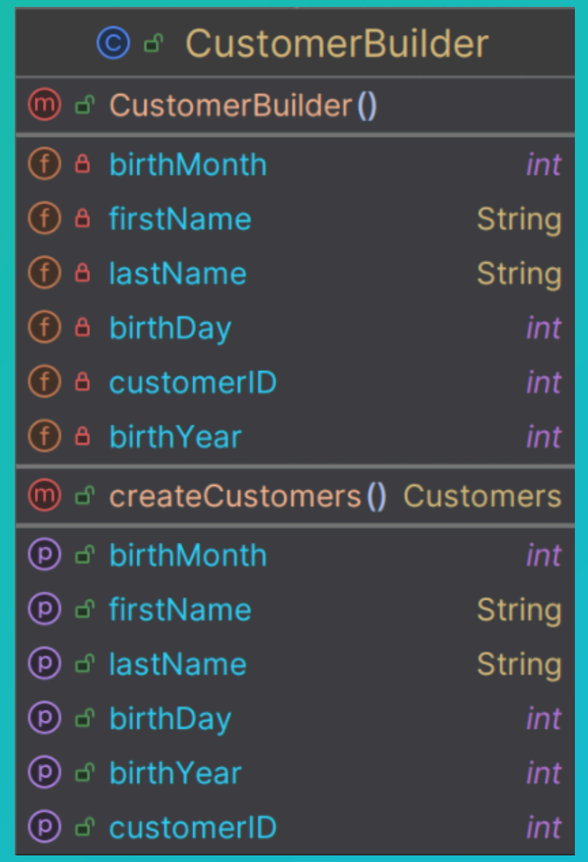


BUILDER DESIGN PATTERN

- The ***Flight*** **class uses the** ***FlightBuilder*** **class to delegate the creation of its objects, enabling a consistent construction process for producing various flight representations.**

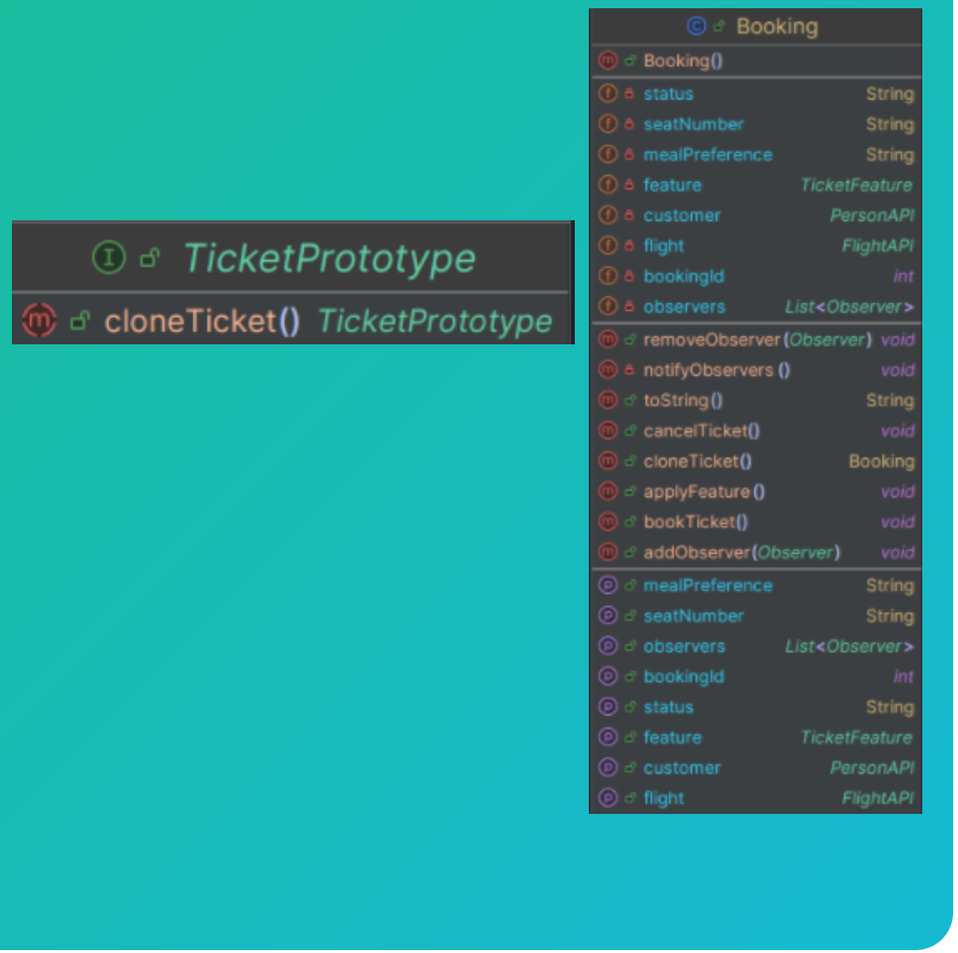
- **This approach simplifies the extension and variation of the internal structures of** ***Flight*** **objects, promoting flexibility and scalability.**

- **The** ***createFlight*****() method is responsible for ultimately creating the** ***Flight*** **object, ensuring encapsulation of the construction logic.**



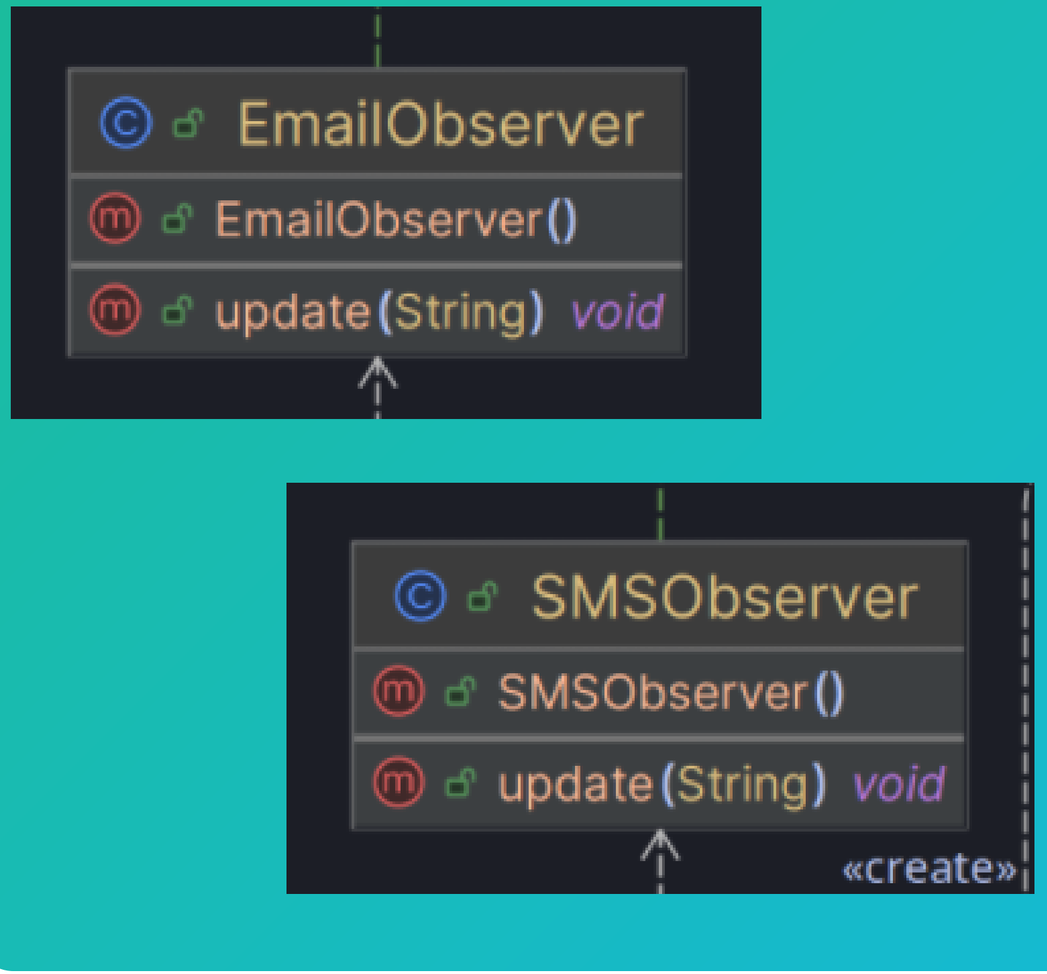
PROTOTYPE DESIGN PATTERN

- **Prototype Interface:** The *TicketPrototype* interface defines the *cloneTicket*() method, ensuring all prototypes implement cloning functionality.
- **Concrete Prototype:** The *Booking* class serves as the concrete prototype, implementing *cloneTicket*() to enable deep copying via *super.clone*().
- **Cloning Mechanism:** The *Client* creates an original *Booking* object, uses *cloneTicket*() to generate a duplicate, and then customizes the cloned instance independently.
- **Key Benefits:** The Prototype pattern simplifies object creation, reduces dependency on constructors, and supports flexible, runtime duplication.



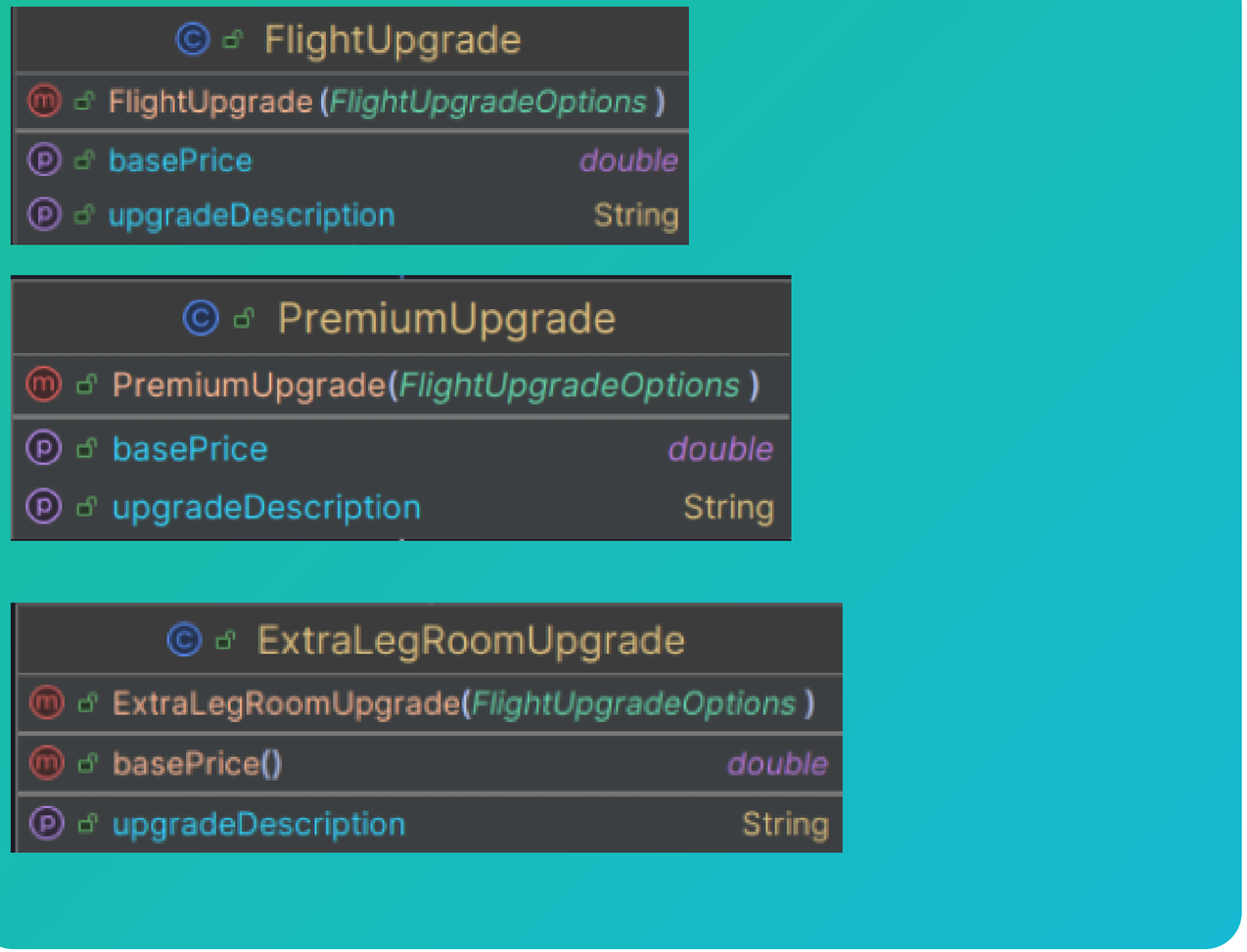
OBSERVER DESIGN PATTERN

- **One-to-Many Dependency:** The Observer pattern establishes a one-to-many relationship, where a subject notifies all registered observers whenever its state changes.
- **Observer Structure:** The pattern includes an abstract *Observer* class, implemented by concrete observers like *EmailObserver* and *SMSObserver*, which define the specific update behaviors.
- **Loose Coupling:** Observers can be added or removed dynamically without altering the subject's code, promoting flexibility and scalability in managing dependencies.



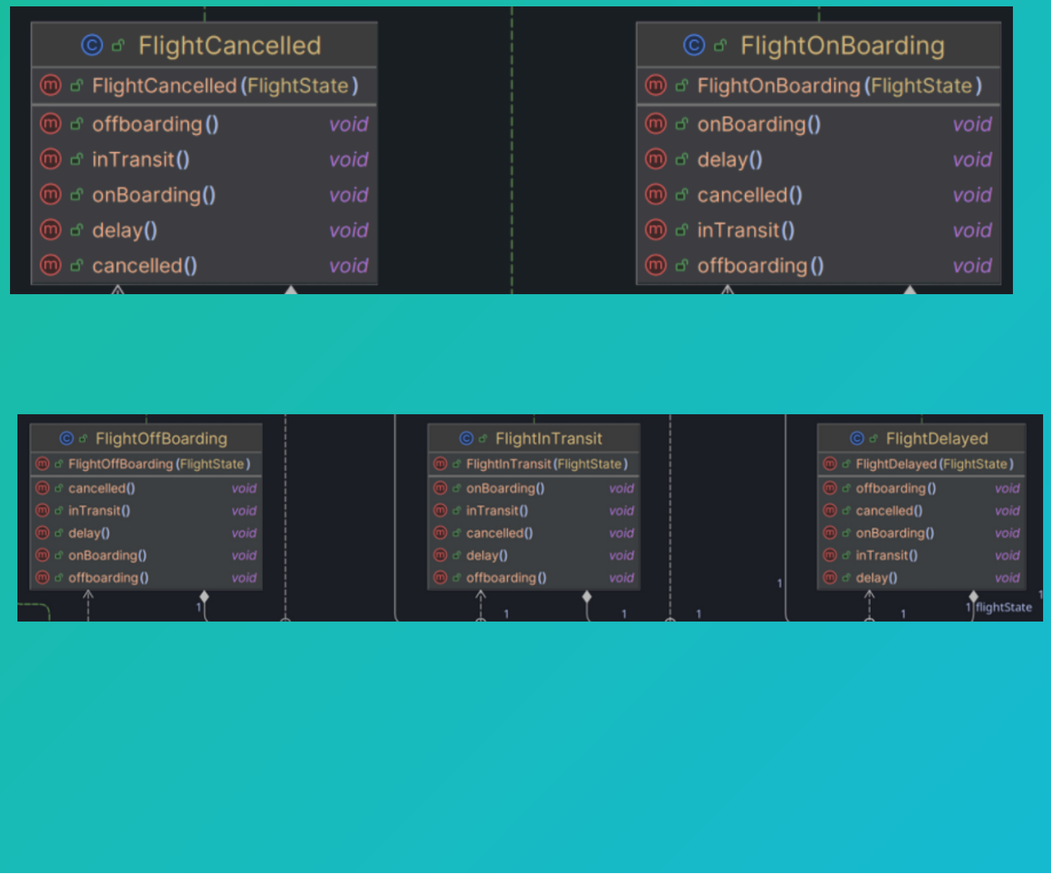
DECORATOR DESIGN PATTERN

- The ***FlightUpgradeOptions*** interface defines methods to be overridden by decorators, enabling flexible enhancements.
- ***FlightUpgrade*** instances hold the base price and description, which decorators modify to apply specific upgrades.
- Implemented decorators include ***ExtraLegRoom***, ***ExtraLuggage***, and ***PremiumUpgrade***, offering various customizable options.



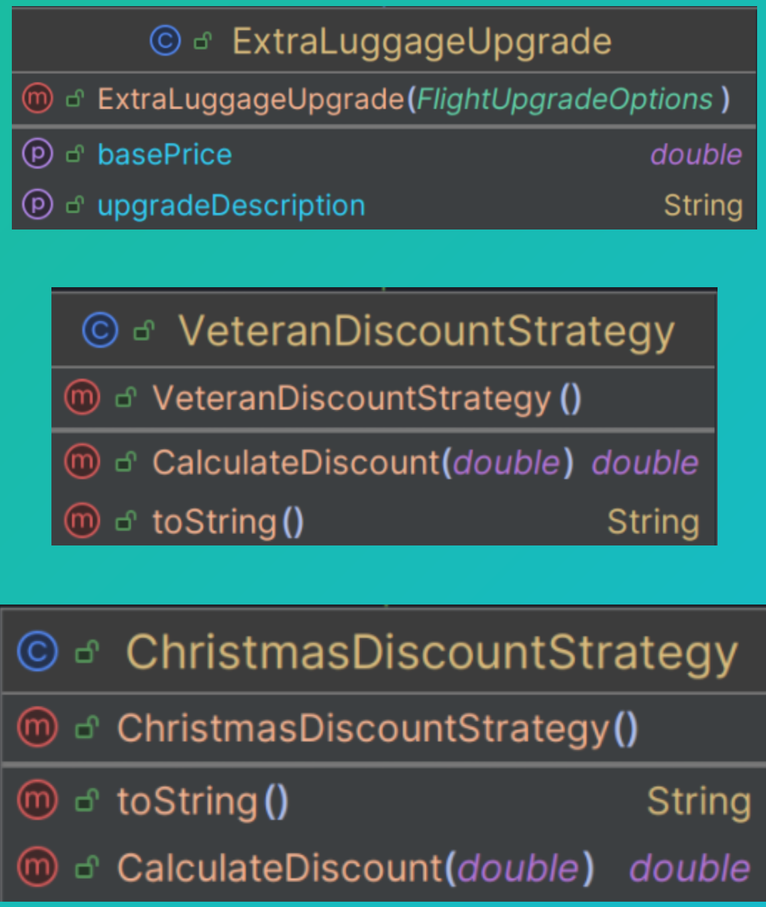
STATE DESIGN PATTERN

- **Dynamic Behavior:** The *Flight* object modifies its behavior dynamically based on its current state, allowing state-specific actions and transitions.
- **State Abstraction:** The *FlightStateAPI* provides an interface for defining the methods that each flight state (e.g., *Scheduled, OnBoarding, InTransit, OffBoarding, Delayed, Cancelled*) must implement.
- **State Flexibility:** The design supports seamless transitions between various states, enabling clear and maintainable management of flight operations.

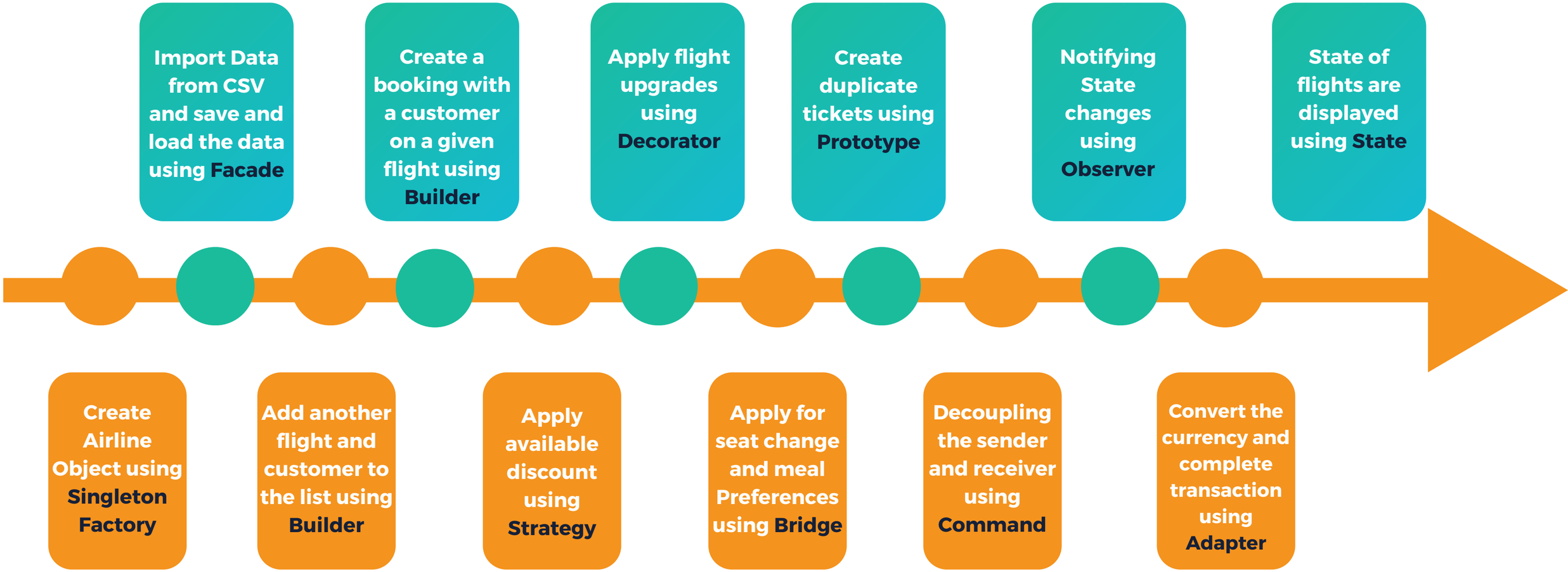


STRATEGY DESIGN PATTERN

- A family of Discount Strategy algorithms is implemented, allowing interchangeable strategies via the ***DiscountStrategyAPI*** for abstraction.
- The ***CalculateDiscount*** function takes the airline price as input, applies the specified discount percentage, and returns the discounted price.
- The strategy can be dynamically modified at runtime, ensuring flexibility and adaptability.



PROGRAM FLOW



FUTURE IDEAS

- **Using design patterns, make the program portable to other forms of travel**
- **Add more robust user and admin authentication**
- **Add a UI with React, and move code to a Spring application**

Group 13

THANKS FOR TEACHING DESIGN PATTERNS.

NISHI PANCHOLI
ANDREA SEQUEIRA
ATHARVA KAMBLE
AZIZ VOHRA
NEHA BALAJI
SATYAJIT DAS