AIRLINE MANAGEMENT SYSTEM

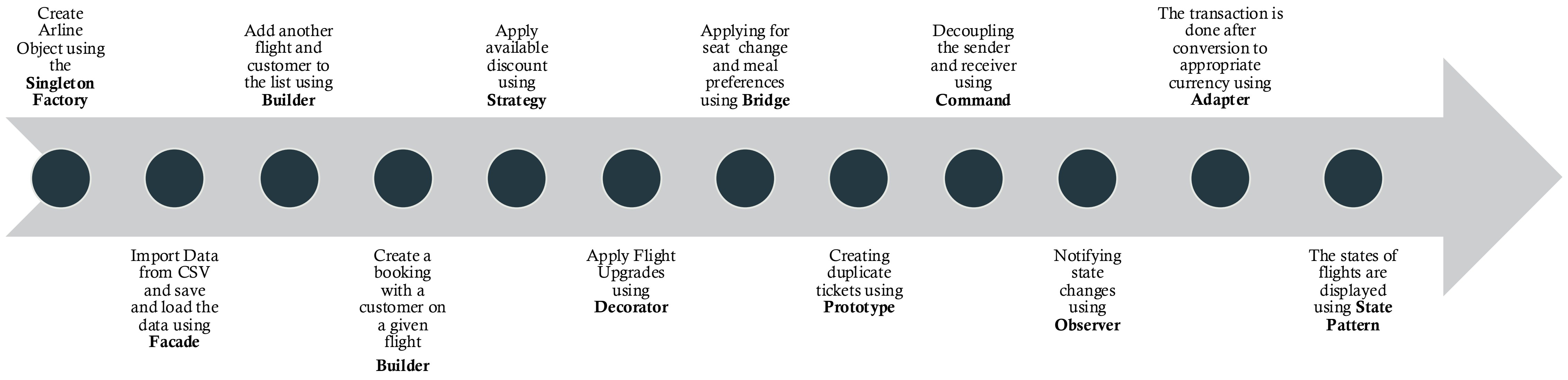
CSYE 7374 - Design Patterns

TEAM MEMBERS

Name	NU ID	Contribution
Rakesh Soni	2813450	Integrated Builder and Decorator patterns for flexible flight and upgrade implementations.
Devansh Srivastava	2272055	Structured core design and implemented Factory and Singleton patterns.
Niraj Komalkant Malpani	2849800	Simplified data workflows using Facade and Bridge patterns for saving/loading and modular features.
Ujjawal Shrivastava	2201611	Enhanced system functionality with Observer and State patterns for dynamic behavior.
Naman Diwan	2724115	Designed and implemented Strategy and Prototype patterns for dynamic discounts and ticket cloning.
Mihir Dadwal	2294525	Implemented Command pattern for modular ticket booking and cancellation features.
Atharva Tiwari	2824041	Developed the Adapter pattern for multi-currency flight price conversions.

DESIGN PATTERNS IMPLEMENTED

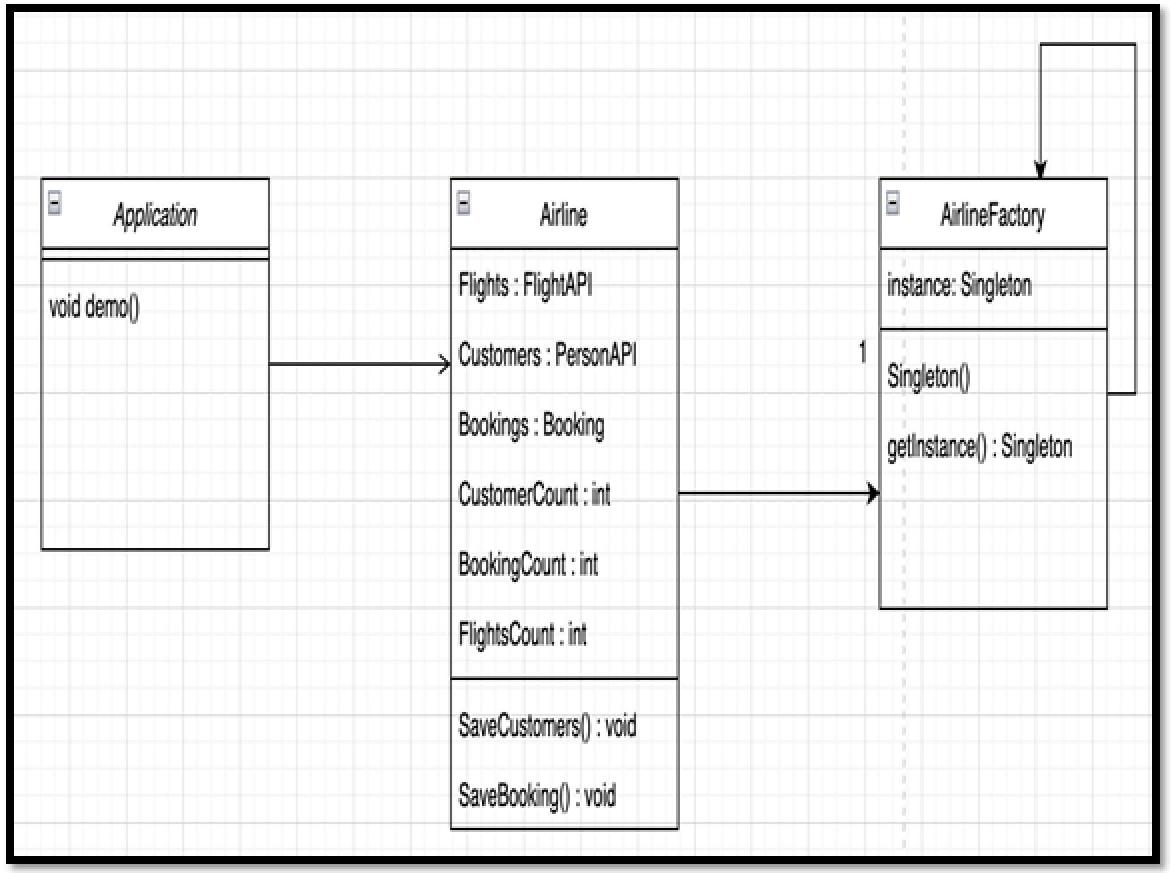
Singleton	Factory	Builder	Facade
Strategy	Decorator	Bridge	Prototype
Command	Observer	Adapter	State


Create Arline Object using the **Singleton Factory**
Import Data from CSV and save and load the data using **Facade**
Add another flight and customer to the list using **Builder**
Create a booking with a customer on a given flight **Builder**
Apply available discount using **Strategy**
Apply Flight Upgrades using **Decorator**
Applying for seat change and meal preferences using **Bridge**
Creating duplicate tickets using **Prototype**
Decoupling the sender and receiver using **Command**
Notifying state changes using **Observer**
The transaction is done after conversion to appropriate currency using **Adapter**
The states of flights are displayed using **State Pattern**

PROGRAM FLOW

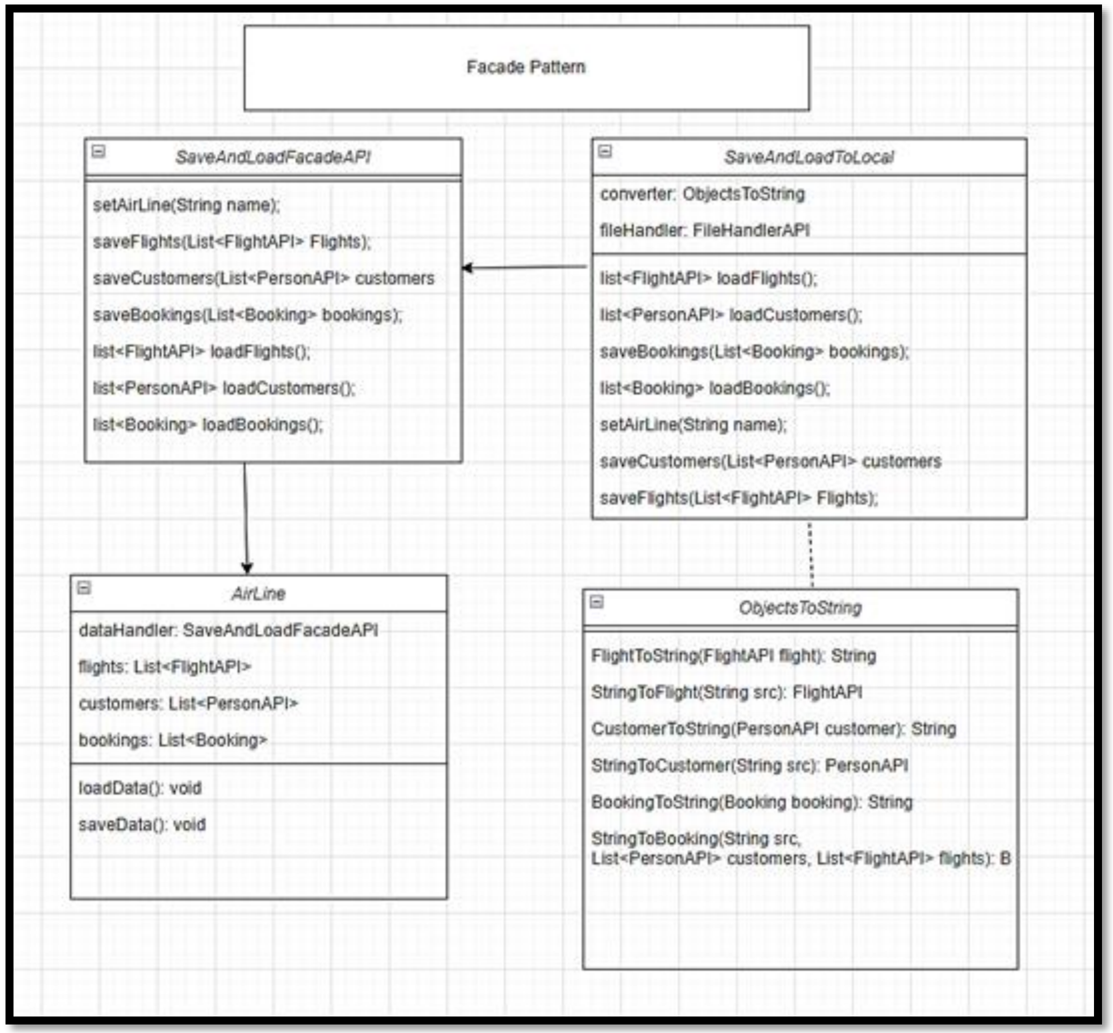
SINGLETON + FACTORY

- The **main** method includes a call to the **demo** method
- Within the **demo** method, an **Airline** object is created, which is provided by **AirlineFactory**
- The **AirlineFactory** uses eager singleton pattern to produce the **Airline** object



FACADE

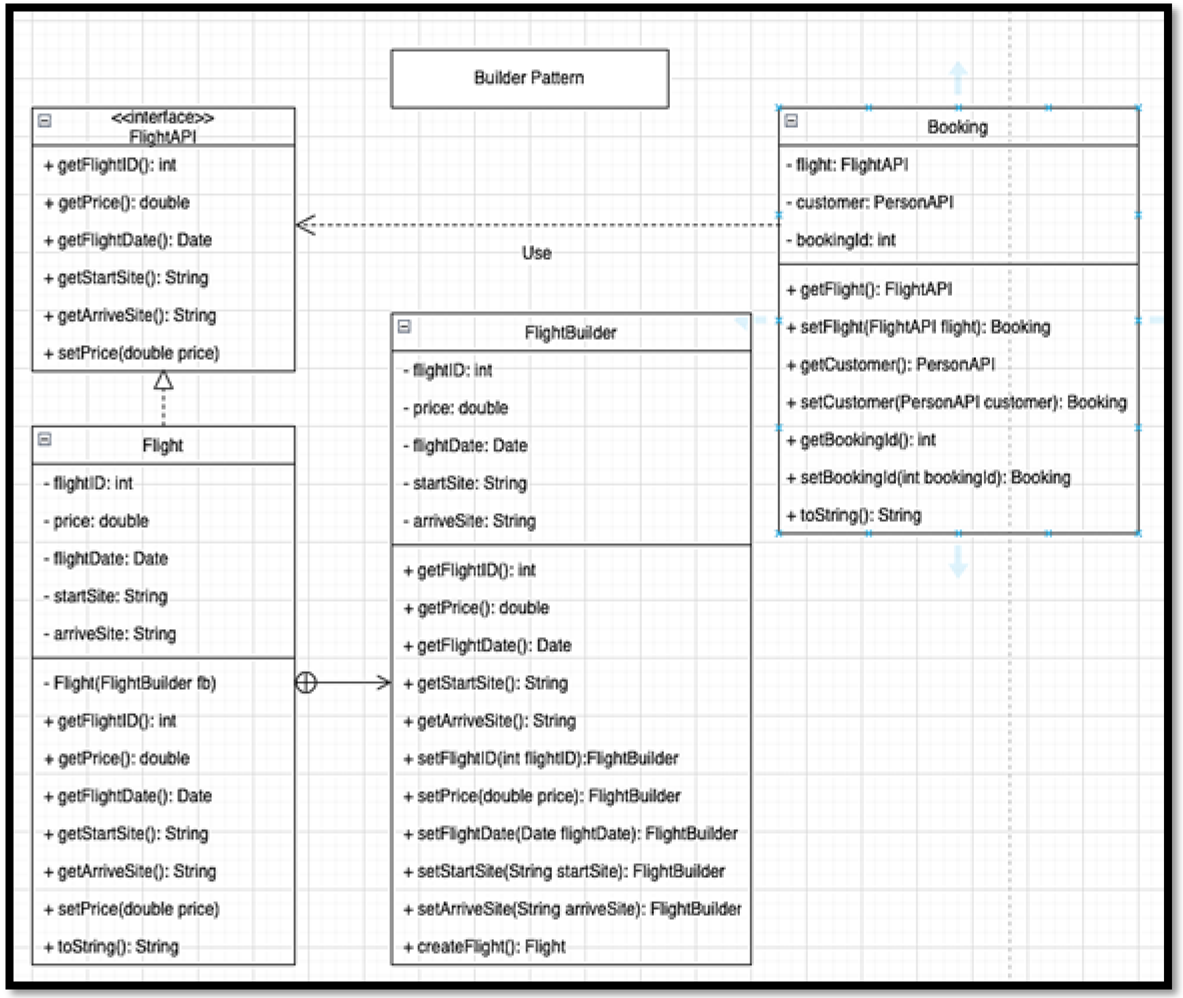
- The complexities of saving and loading data when the program starts or stops are hidden from the user
- Various methods of saving and loading data can be easily implemented by using the **SaveAndLoadFacadeAPI**
- The project includes an implementation that allows saving and loading data into CSV files
- An instance of the **SaveAndLoadFacadeAPI** implementation is integrated in **AirLine** object


Facade Pattern
SaveAndLoadFacadeAPI
setAirLine(String name);
saveFlights(List<FlightAPI> Flights);
saveCustomers(List<PersonAPI> customers
saveBookings(List<Booking> bookings);
list<FlightAPI> loadFlights();
list<PersonAPI> loadCustomers();
list<Booking> loadBookings();
SaveAndLoadToLocal
converter: ObjectsToString
fileHandler: FileHandlerAPI
list<FlightAPI> loadFlights();
list<PersonAPI> loadCustomers();
saveBookings(List<Booking> bookings);
list<Booking> loadBookings();
setAirLine(String name);
saveCustomers(List<PersonAPI> customers
saveFlights(List<FlightAPI> Flights);
AirLine
dataHandler: SaveAndLoadFacadeAPI
flights: List<FlightAPI>
customers: List<PersonAPI>
bookings: List<Booking>
loadData(): void
saveData(): void
ObjectsToString
FlightToString(FlightAPI flight): String
StringToFlight(String src): FlightAPI
CustomerToString(PersonAPI customer): String
StringToCustomer(String src): PersonAPI
BookingToString(Booking booking): String
StringToBooking(String src, List<PersonAPI> customers, List<FlightAPI> flights): B

BUILDER

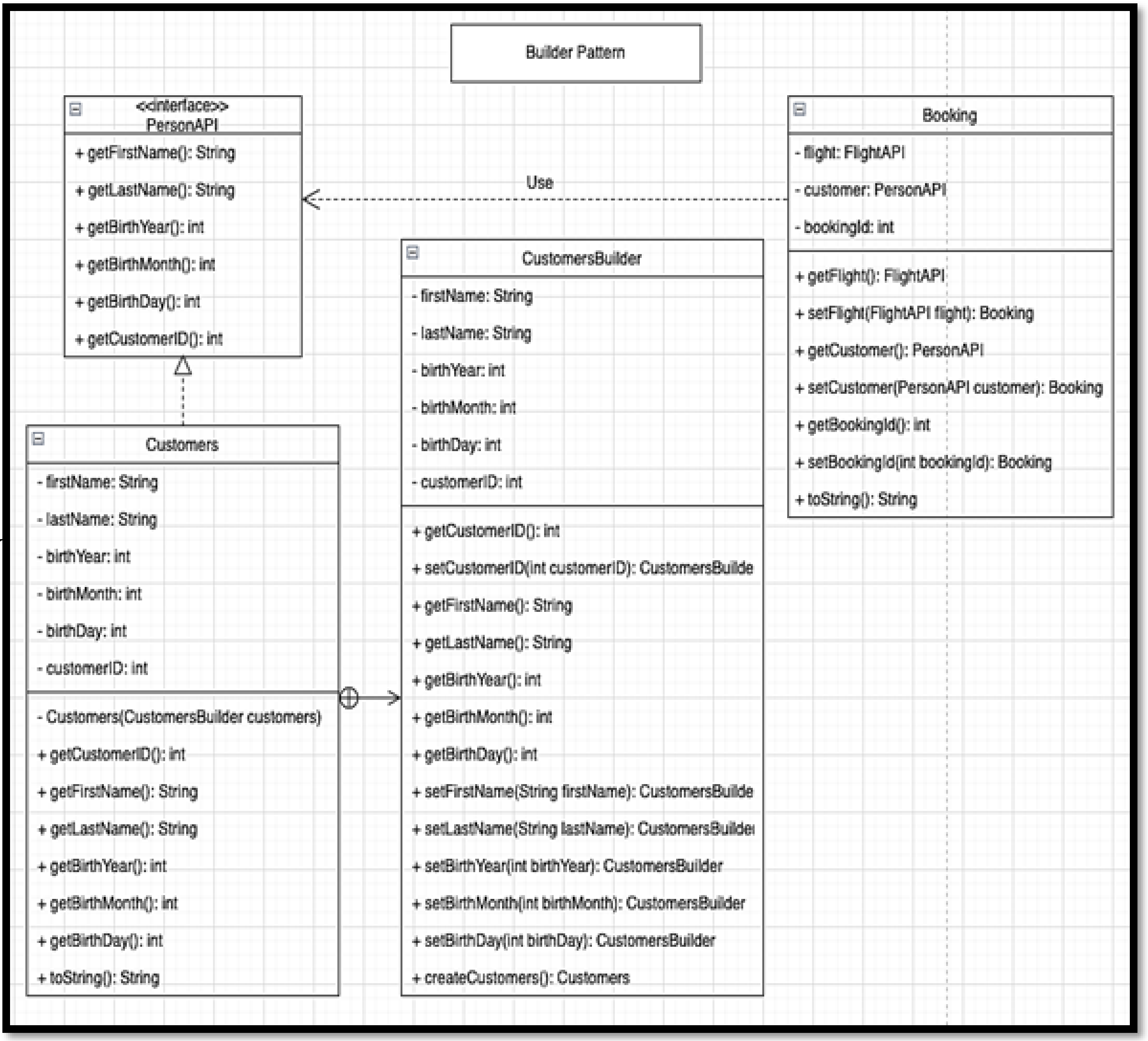
- The **Flight** class delegates the creation of its objects to the **FlightBuilder** class. This allows the same construction process to produce various representations, simplifying the extension and variation of the internal structures of flights.

- The **Flight** object is ultimately created by the **createFlight()** method



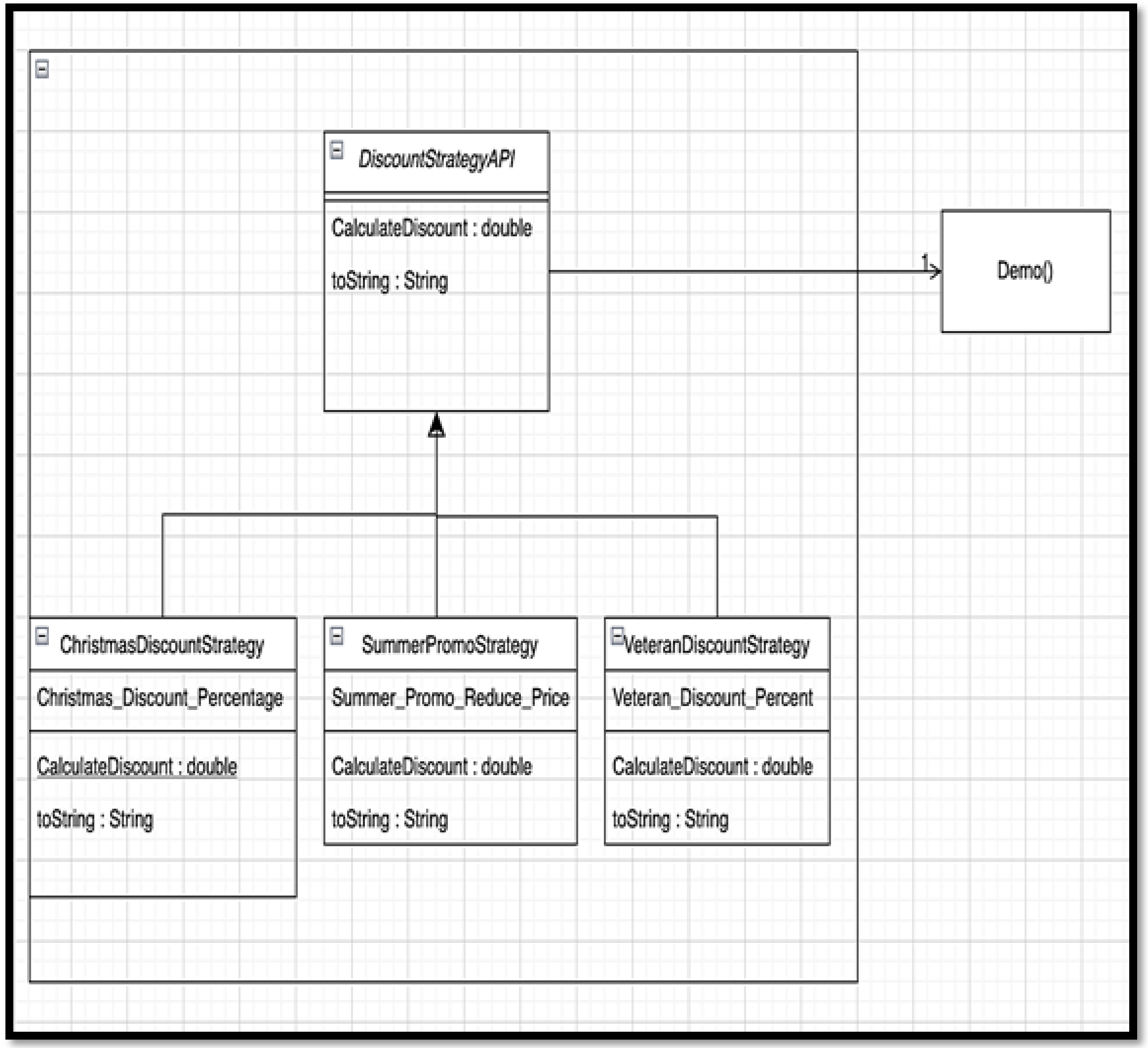
BUILDER

- The **PersonAPI** provides an interface for various classes to implement. If the Observable pattern is introduced later, the **Subscribers** class can also implement **PersonAPI**
- The **Customers** class hands over the responsibility of object creation to the **CustomersBuilder** class, which aids in extending and diversifying the internal representations of customers
- The **Customers** object is ultimately created using the **createCustomers** method



STRATEGY

- The family of Discount Strategy Algorithms are interchangeable and the **DiscountStrategyAPI** specified by super class for abstraction
- The airline price is passed as an argument to the **CalculateDiscount** function, which applies the specified percentage and returns the discounted price
- The strategy can be modified during runtime.


DiscountStrategyAPI
CalculateDiscount : double
toString : String
1
Demo()
ChristmasDiscountStrategy
Christmas_Discount_Percentage
CalculateDiscount : double
toString : String
SummerPromoStrategy
Summer_Promo_Reduce_Price
CalculateDiscount : double
toString : String
VeteranDiscountStrategy
Veteran_Discount_Percent
CalculateDiscount : double
toString : String

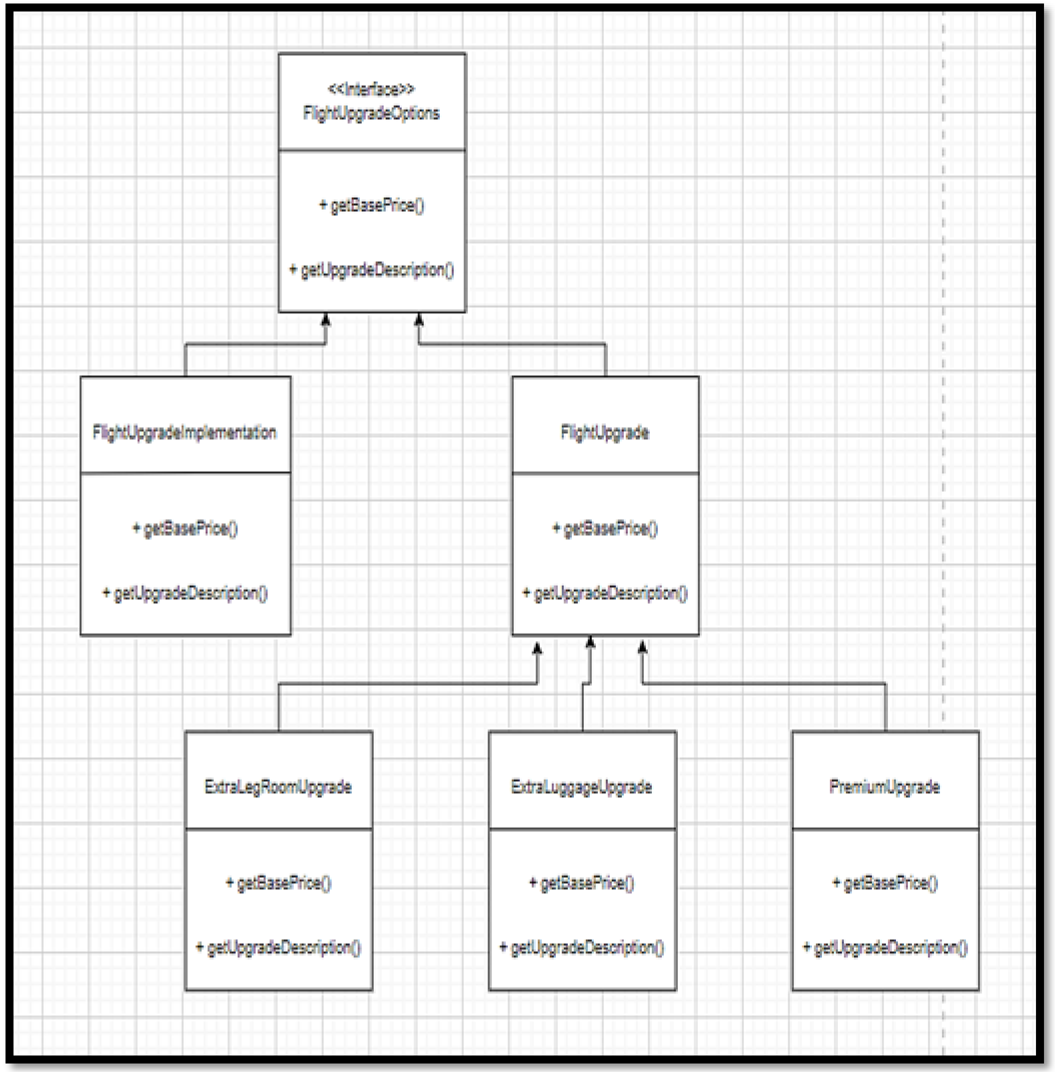
DECORATOR

- The **FlightUpgradeOptions** interface is used to define methods that decorators will override when implementing it

- An instance of **FlightUpgrade** contains the base price and description, which are passed to the decorator methods for modifications

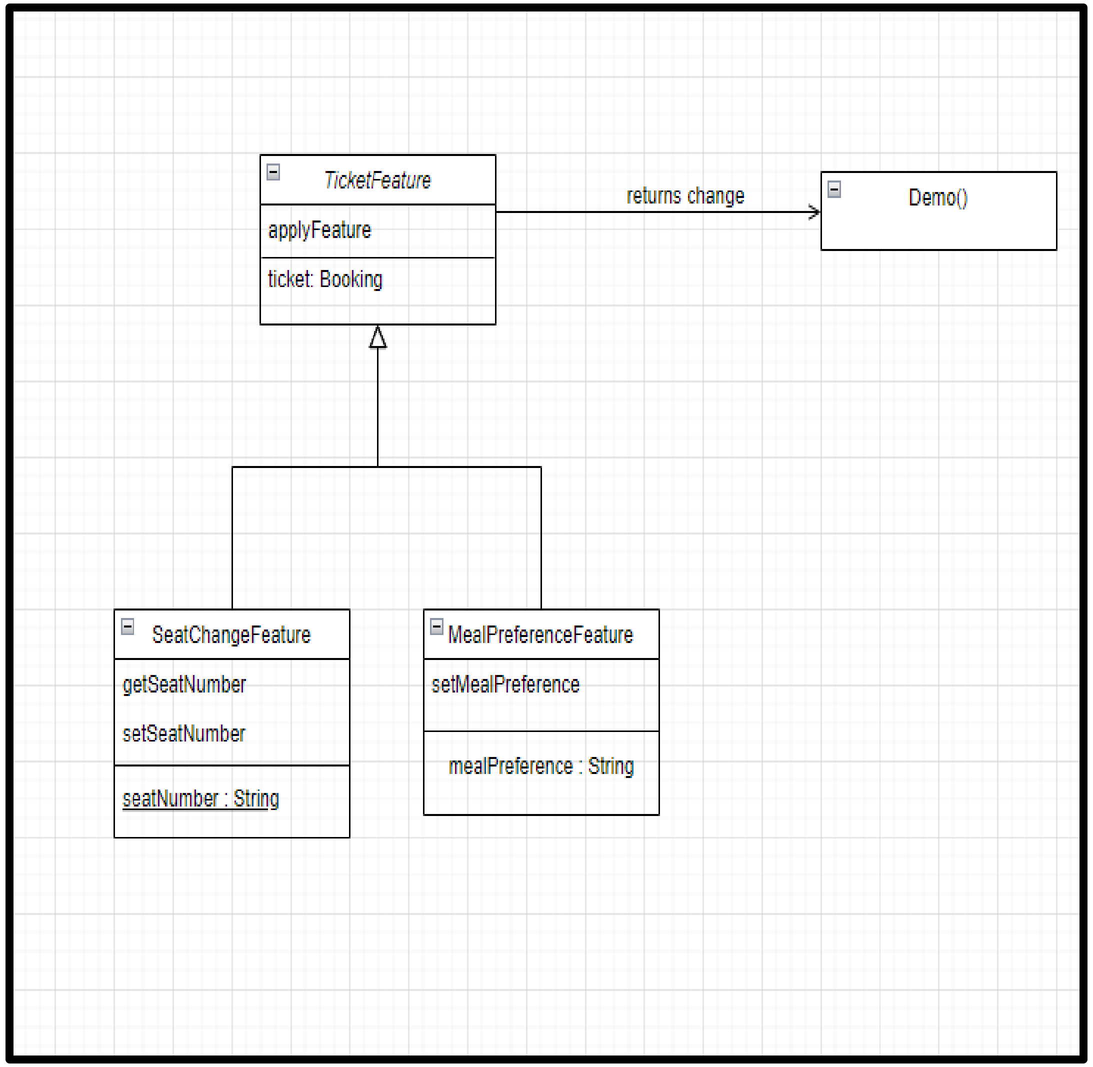
- **FlightUpgradeImplementation** retrives the base price and description

- Decorators include:
– ExtraLegRoom
– ExtraLuggage
– PremiumUpgrade



BRIDGE

- The Bridge pattern separates abstraction (TicketFeature) from implementation (SeatChangeFeature and MealPreferenceFeature).
- It allows for flexibility by enabling independent variation of abstraction and implementation hierarchies.
- The pattern facilitates extending functionality without modifying existing code, as seen with the Demo class interaction.



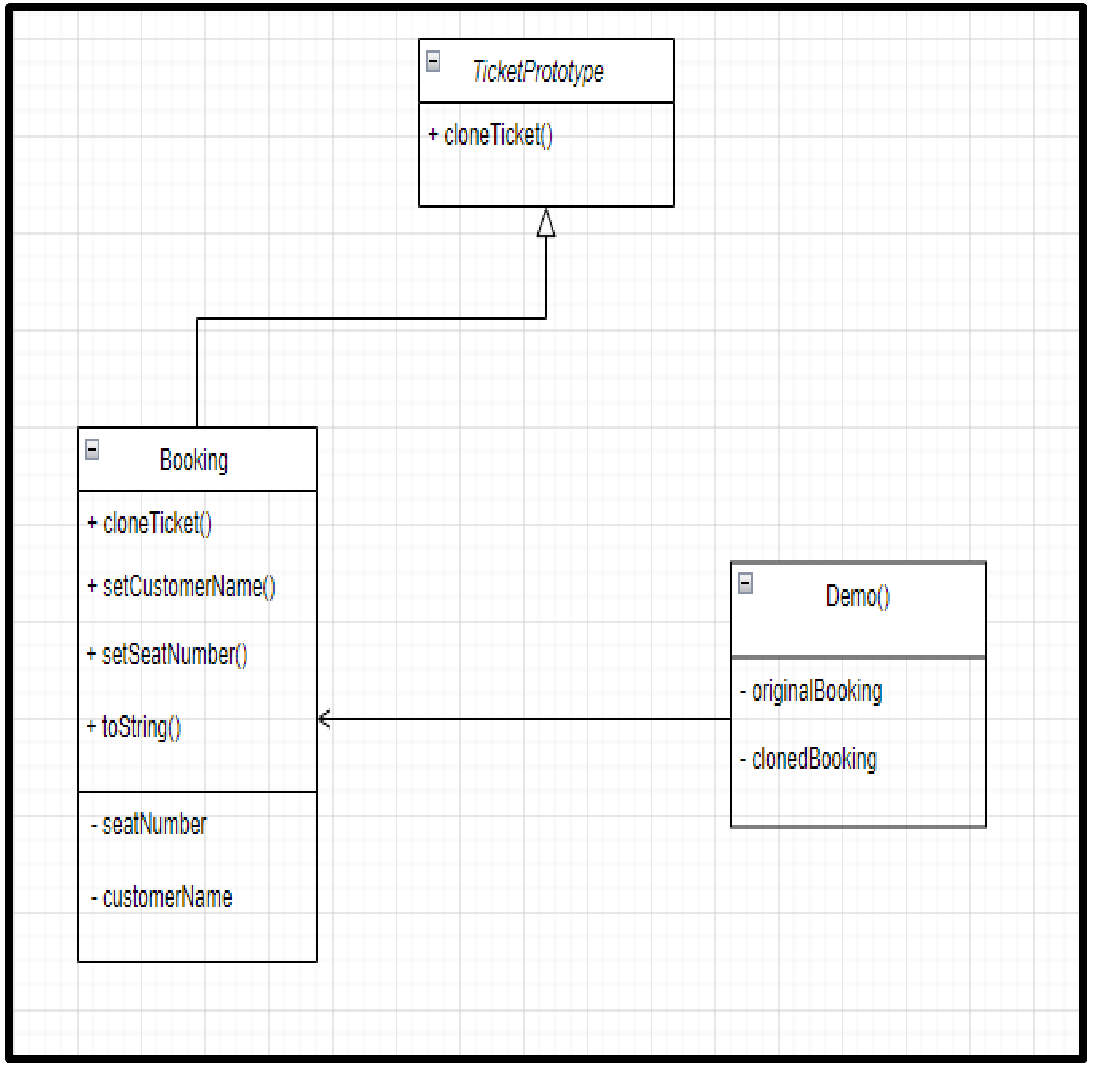
PROTOTYPE

- TicketPrototype interface defines the cloneTicket() method that concrete prototype classes will implement
- An instance of Booking serves as the concrete prototype, implementing cloneTicket() to provide a copy of itself.

The Booking class includes:

- cloneTicket(): Implements cloning by calling super.clone().

Client creates an original Booking object, clones it using cloneTicket(), and modifies the cloned instance.

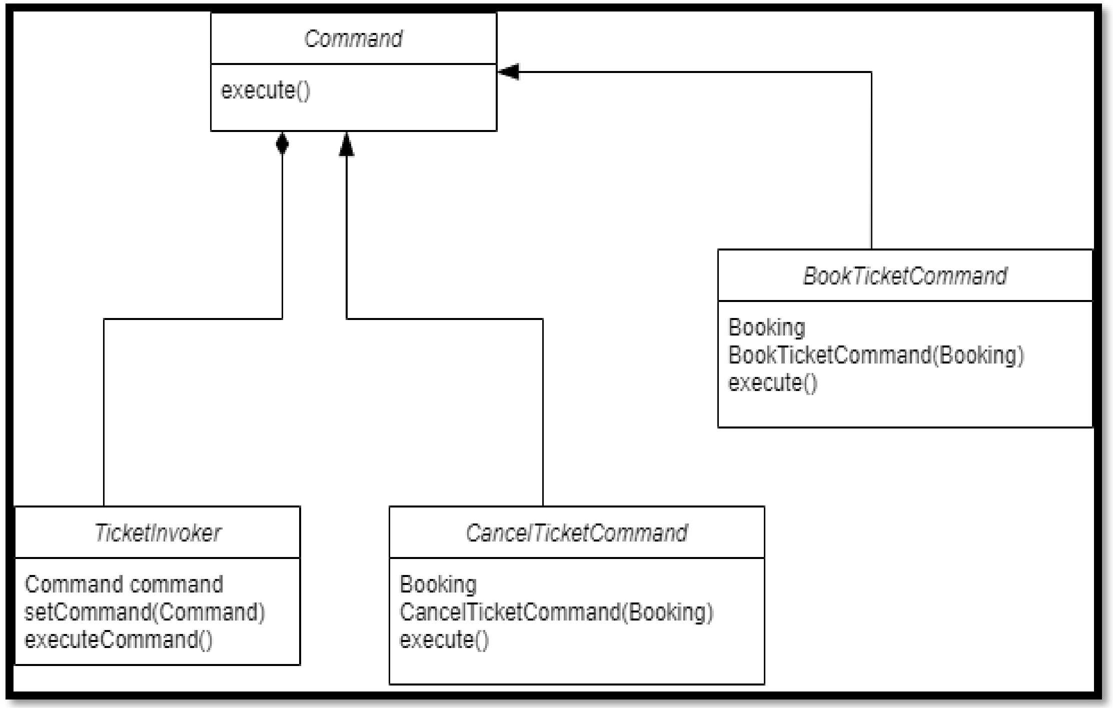


COMMAND

- Command interface defines the execute() method that concrete command classes will implement.
- Instance of Booking contains methods bookTicket() and cancelTicket() which are invoked by command objects

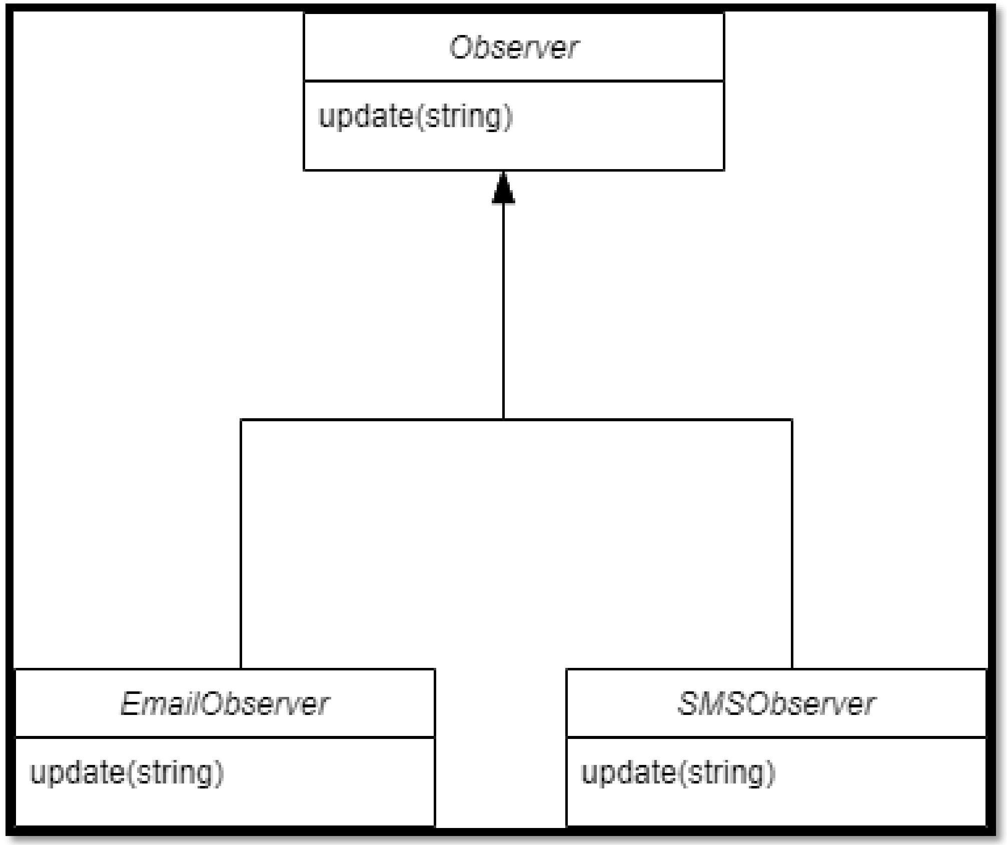
Concrete command classes include:
- BookTicketCommand: invokes bookTicket()
- CancelTicketCommand: invokes cancelTicket()

Client creates instances of BookTicketCommand and CancelTicketCommand, sets them in the TicketInvoker



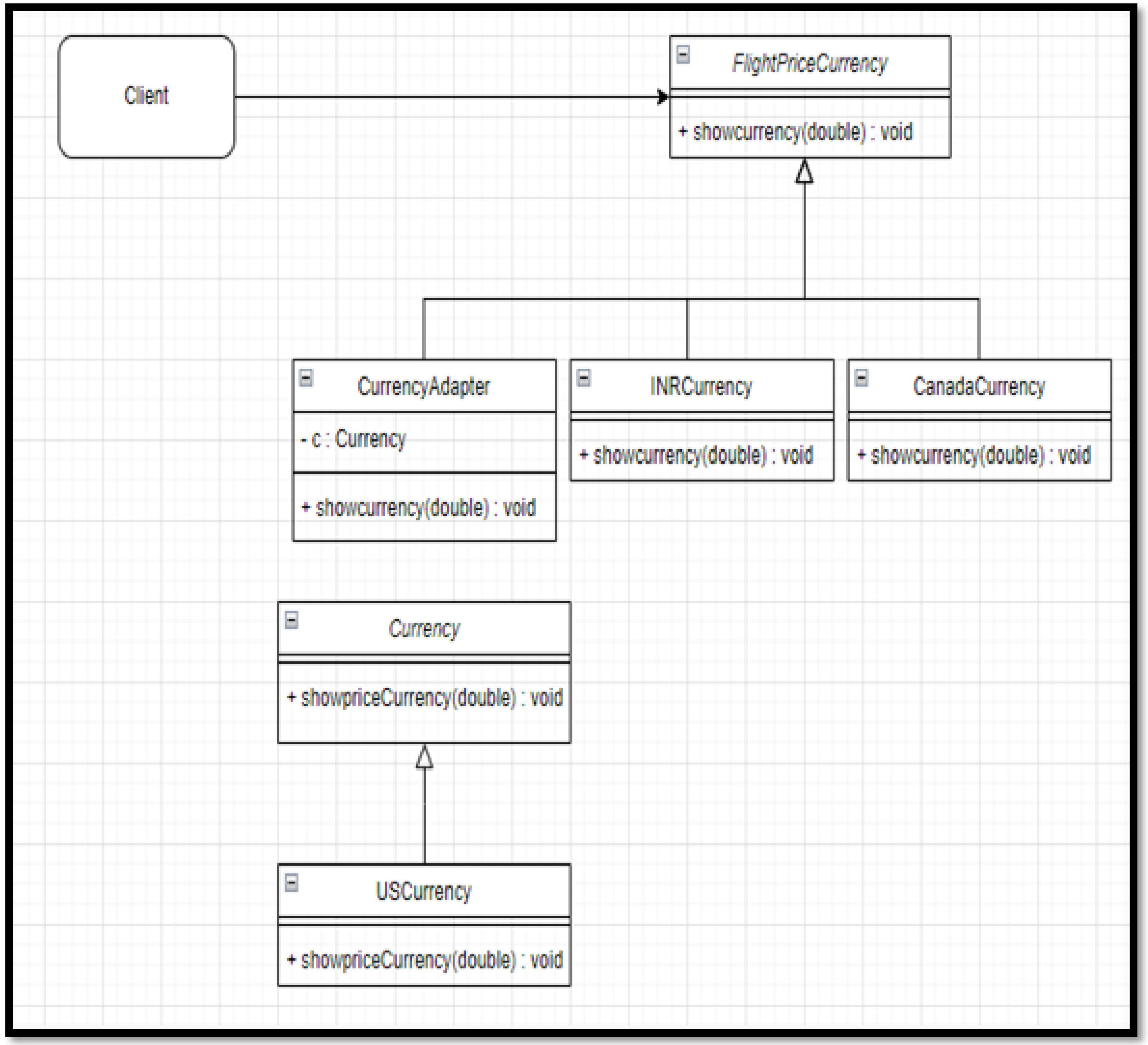
OBSERVER

- The Observer pattern defines a one-to-many dependency between objects, where multiple observers are notified of changes in a subject's state.
- It consists of an abstract Observer class and concrete observer implementations (EmailObserver and SMSObserver in this case).
- The pattern allows for loose coupling between the subject and its observers, enabling easy addition or removal of observers without modifying the subject.



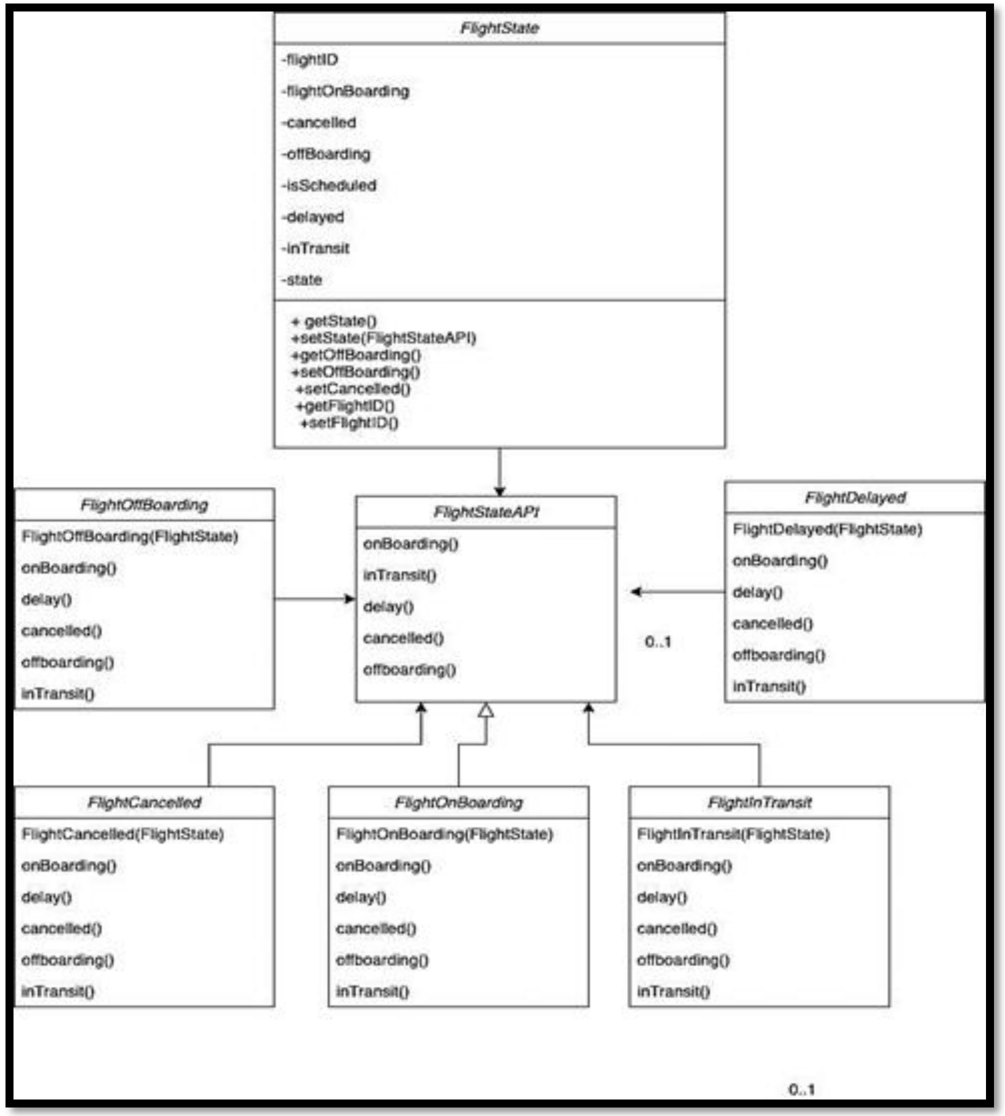
ADAPTER

- The client can view flight prices in various currencies, including INR, Canadian Dollars or US Dollars
- The **CurrencyAdapter** contains an instance of the **Currency** interface to adapt the currency
- The **CurrencyAdapter** conceals the use of the adapted (Legacy API) code that provides flight prices in US dollars


Client
FlightPriceCurrency
+ showcurrency(double) : void
CurrencyAdapter
- c : Currency
+ showcurrency(double) : void
INRCurrency
+ showcurrency(double) : void
CanadaCurrency
+ showcurrency(double) : void
Currency
+ showpriceCurrency(double) : void
USCurrency
+ showpriceCurrency(double) : void

STATE

- The **Flight** state object alters its behavior depending on its current internal state
- The **FlightStateAPI** defines the methods that each state of the flight must implement
- The possible states of the flight include: Scheduled, OnBoarding, InTransit, OffBoarding, Delayed, Cancelled



FUTURE ENHANCEMENTS AND LESSONS LEARNED

Future Enhancements:

- **Real-Time Data Sync**: Integrate with live APIs for flight availability and pricing.
- **Database Integration**: Replace CSV with a robust database for better scalability.
- **Mobile Support**: Build a mobile application for customer convenience.
- **Enhanced UX/UI**: Develop a graphical interface for better user interaction.
- **Multi-Language Support**: Offer localization for international customers.

Lessons Learned:

- **Team Collaboration**: Effective communication is critical for modular development.
- **Design Principles**: Using design patterns simplifies complex systems.
- **Testing and Debugging**: Importance of unit testing for reliability.
- **Flexibility**: Patterns like Strategy and Observer enhance system adaptability.
- **Version Control**: Git workflows streamlined teamwork and conflict resolution.

THANK YOU

ANY QUESTIONS/COMMENTS/CONCERNS?