WS2019/20 – Design Patterns and Frameworks

Architecture Mismatch Patterns

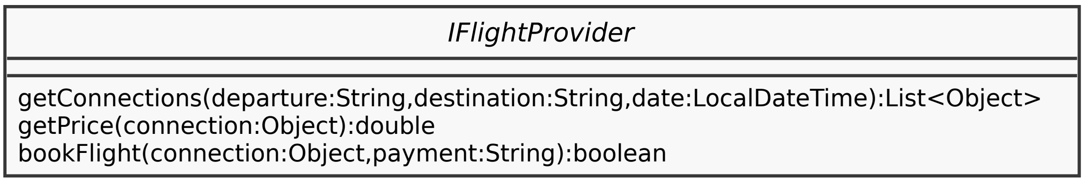
Professor: Prof. Dr. Uwe Aßmann
Lectuer: Dr.-Ing. Sebastian Götz
Tutor: Dr. rer. nat. Marvin Triebel

Task 1 Medi(t)ative Air

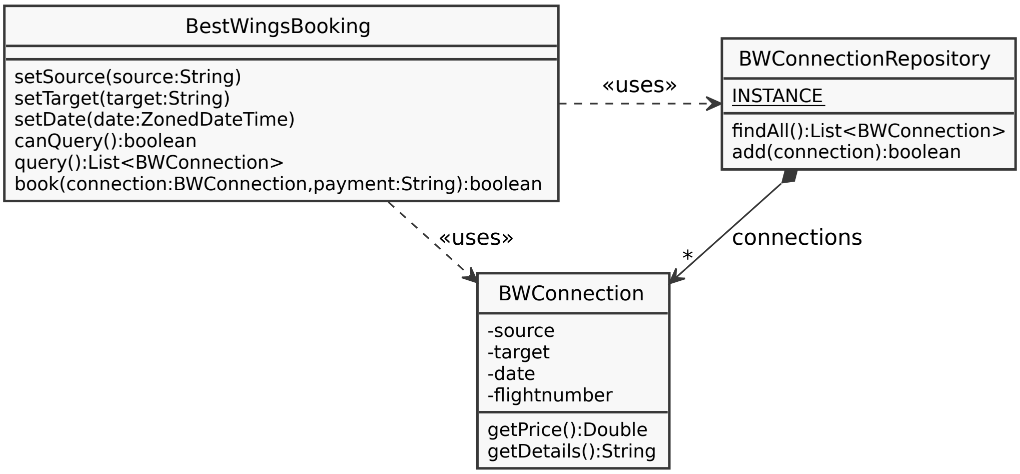
This exercise focuses on specific *Design Patterns* that allow for connecting possibly incompatible classes and structures [1].

In this task a flight booking service should be designed, which enables querying for the cheapest flight to a destination of your choice out of a number of providers, as well as booking a selected flight. For the sake of simplicity, assume that each *airline* provides their own proprietary class, which provides operations for querying for connections and booking a flight. To simplify the integration of the different proprietary classes your application provides an IFlightProvider interface, which encompasses the following generic methods:

- getConnections(departure,destination,date):List<Connection> permits querying all connections from an airport of departure to a destination airport at a given date (around the given time).
- getPrice(connection):double returns the price of the given connection in Euro or throw an exception if the connection was invalid.
- bookFlight(connection,payment):boolean permits booking a given connection providing the payment details. This method indicates its success returning true, whereas its failure is indicated with an exception.

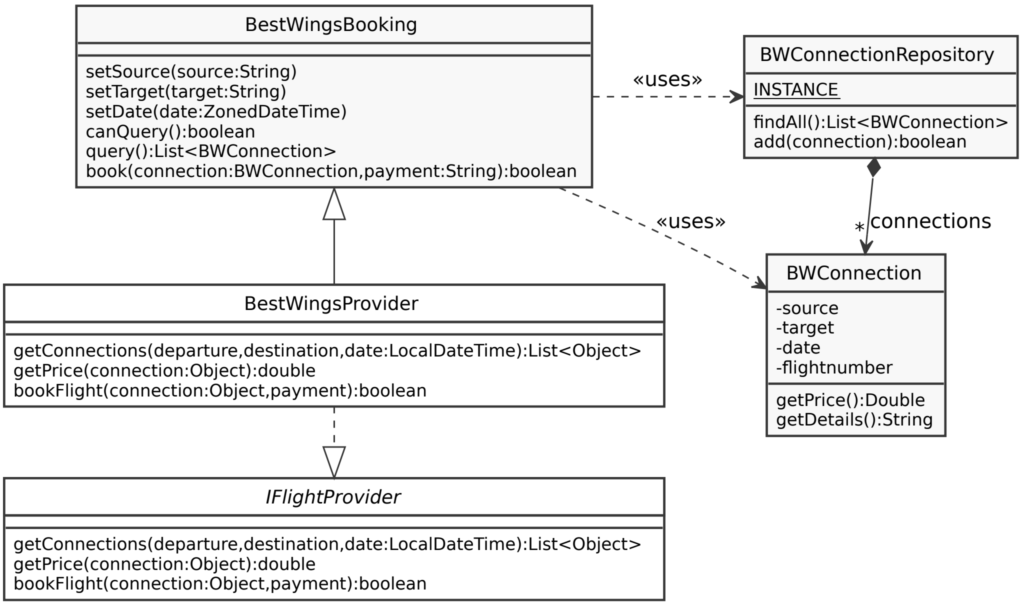


a) Based on the `IFlightProvider` interface, the first task is to design and implement a `Class Adapter` for the class provided by *Best Wings*, as depicted below:



Solution: To apply the `Class Adapter` design pattern, a subclass of the adapted class`BestWingsBooking` has to be created, which implements the `IFlightProvider` interface. This solution is feasible, due to two reasons. First, `BestWingsBooking` has no direct references to other classes, as it only has implicit dependencies to the `Singleton` `BWConnectionRepository` and `BWConnection`. Second, the class `BestWingsBooking` already provides services that are very similar to those required by the interface `IFlightProvider`.

Note an `Object Adapter` would also be suitable.

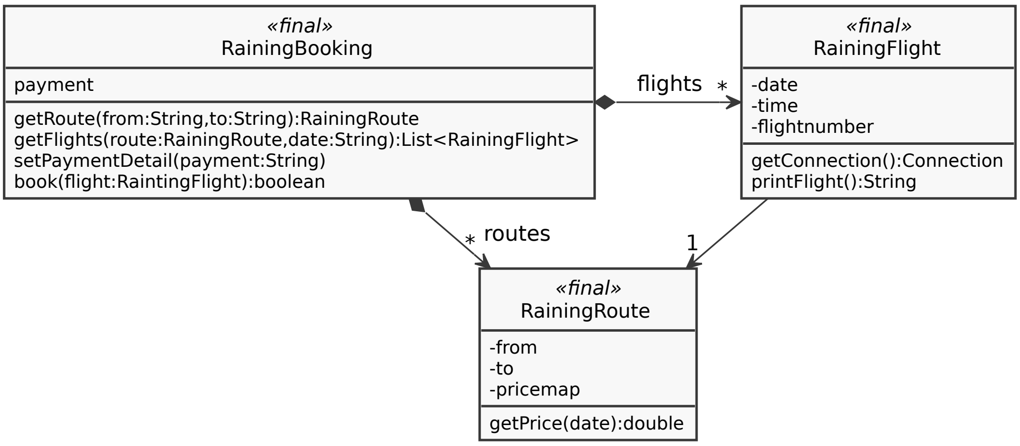


Following this design the `BestWingsProvider` is implemented below.

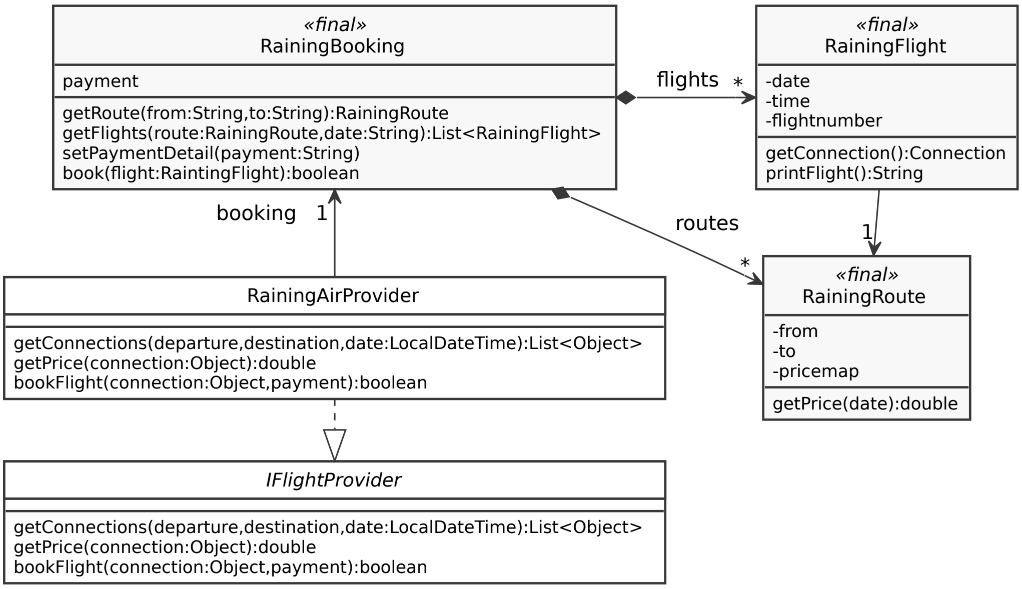
Listing 1: Implementation of the BestWingsProvider class.

```java
public class BestWingsProvider extends BestWingsBooking implements IFlightProvider{
  public List<Object> getConnections(String departure, String destination,LocalDateTime date){
    setSource(departure);
    setTarget(destination);
    setDate(date.toLocalDateTime());
    if (canQuery())
      return query();
    else
      return Collections.emptyList();
  }
  public boolean bookFlight(Object connection, String payment){
    if (! connection instanceof BWConnection)
      return false;
    return book((BWConnection) connection,payment);
  }
  public double getPrice(Object connection){/*...*/}
}
```

b) The next task is be to design and implement an `Object Adapter` to the interface `IFlightProvider` for the `final` class provided by *Raining Air*, shown below:



Solution: To apply the `Object Adapter` design pattern, the `RainingAirProvider` is created as a new class, which implements the `IFlightProvider` interface and has a reference to a `RainingBooking` instance. This design is the only option, due to two reasons. First, the class is marked as <<final>> and cannot be inherited from. However, even if we could subclass it, the `RainingBooking` class still has two composite relations that are filled by the framework upon instantiation. Hence, the adapter class would lake the filled relations, if it is not instantiated by the *Raining Air* framework. Consequently, the `Object Adapter` is the only viable solution.



The corresponding implementation of the `RainingAirProvider` is sketched below.

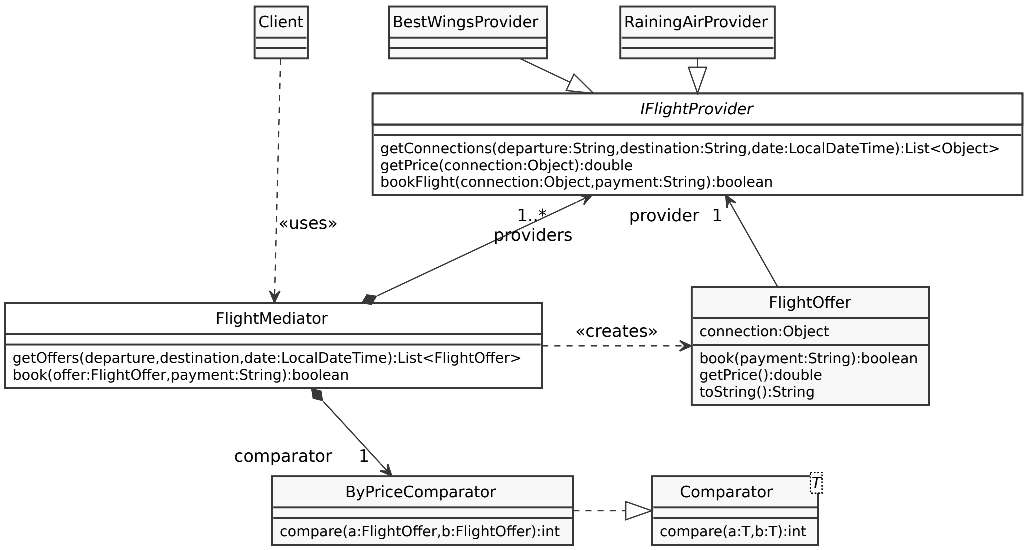
Listing 2: Implementation of the RainingAirProvider class.

```java
public class RainingAirProvider implements IFlightProvider{
  private String format=/*...*/
  private RainingBooking booking;
  public RainingAirProvider(RainingBooking booking){
    this.booking=booking;
  }
  public List<Object> getConnections(String departure, String
      destination,LocalDateTime date){
    RainingRoute r=booking.getRoute(departure,destination);
    return booking.getFlights(r,date.format(formatter));
  }
  public boolean bookFlight(Object connection, String payment){
    if (! connection instanceof RainingFlight)
      return false;
    booking.setPaymentDetail(payment);
    return booking.book((RainingFlight) connection);
  }
  public double getPrice(Object connection){/*...*/}
}
```

c) Now the flight booking service can be designed, which enables clients to query for the cheapest flights and book the preferred flight independent of the airline providing it. Moreover, airlines should not require (and receive) any knowledge from the flights of other airplanes on other flight providers known to the system. Furthermore, the number and implementations details for each airline should be hidden from the users and clients of your application.

Which design pattern could be used? Apply this pattern to design and implement the flight booking service.

Solution: The `Mediator` design pattern would be a good choice, as it establishes a common interface to the services of the different flight providers, i.e., querying and booking, while hiding their implementation details.



Following the `Mediator` design pattern the implementation of the `FlightMediator` is shown below.

Listing 3: Implementation of the FlightMediator class.

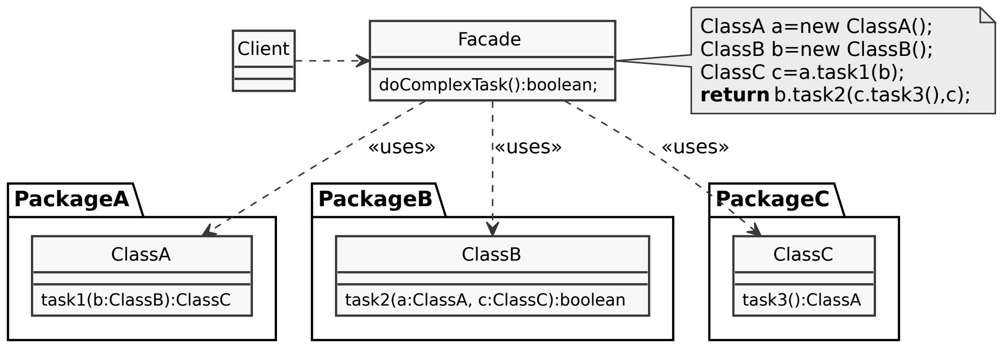
```java
public class FlightMediator{
  private final Comparator<FlightOffer> comparator=
    new ByPriceComparator();
  public List<FlightOffer> getOffers(String departure, String
      destination, LocalDateTime date){
    List<FlightOffer> result=new ArrayList<>();
    for (IFlightProvider p:providers)
      for (Object c:p.getConnections(departure,destiantion,date))
        result.add(new FlightOffer(c,p));
    Collections.sort(result,comparator);
    return result;
  }
  public boolean book(FlightOffer offer, String payment){
    return offer.book(payment);
  }
}
```

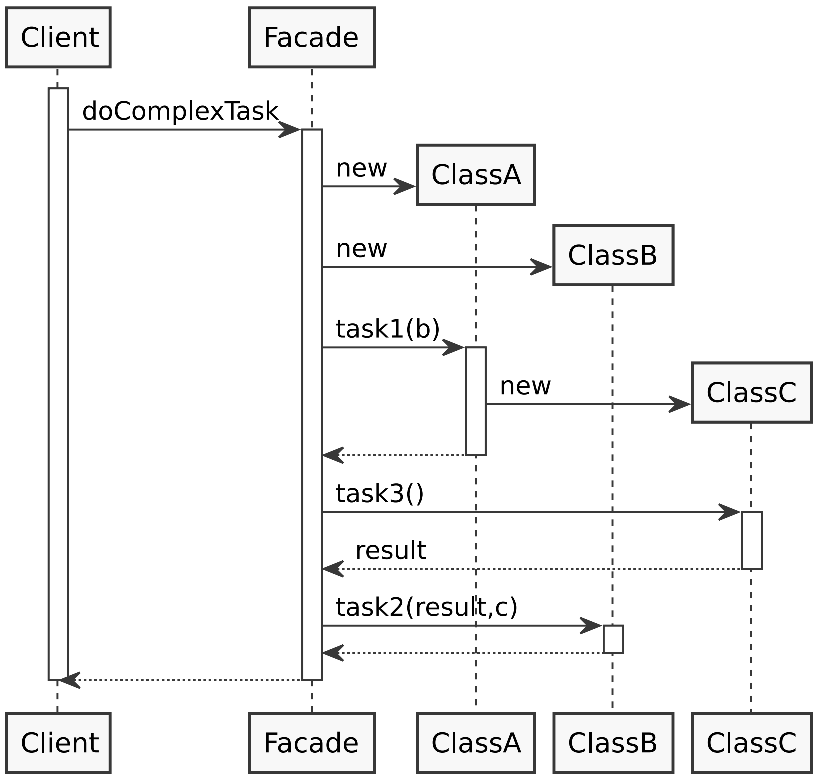
Task 2 Facade Travel Agency

After introducing the flight booking service, this task extends it to a full travel booking service, which permits planning and booking the whole trip from selecting the cheapest flight to picking the best hotel. For simplicity, it is assumed that there exist a similar hotel booking service, i.e., a class for querying and booking hotels. However, the clients should only need to interact with one service class.

a) Which design pattern should be employed? What is its structure?

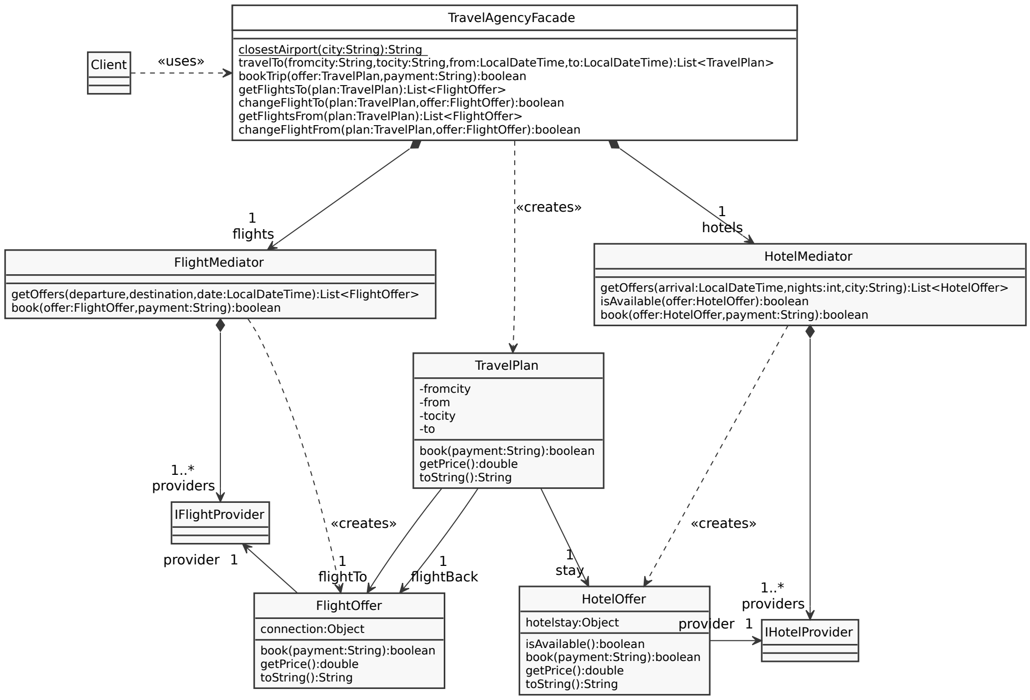
Solution: The `Facade` design pattern [1], depicted below, could be employed, as it hides the complex interactions of multiple objects.





b) Design and implement the travel booking service.

Solution: Accordingly, the faced of the *Travel Agency* encapsulated the querying and booking of the flight to, the flight back, as well as the hotel. The resulting structure is depicted below.



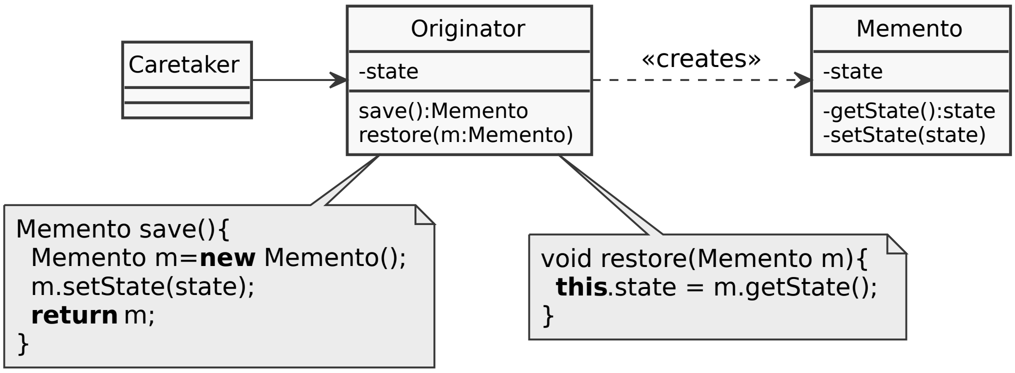
The complex process of assembling `TravelPlans` to a given destination city is implemented into the `travelTo` method shown below.

Listing 4: Implementation of the travelTo method.

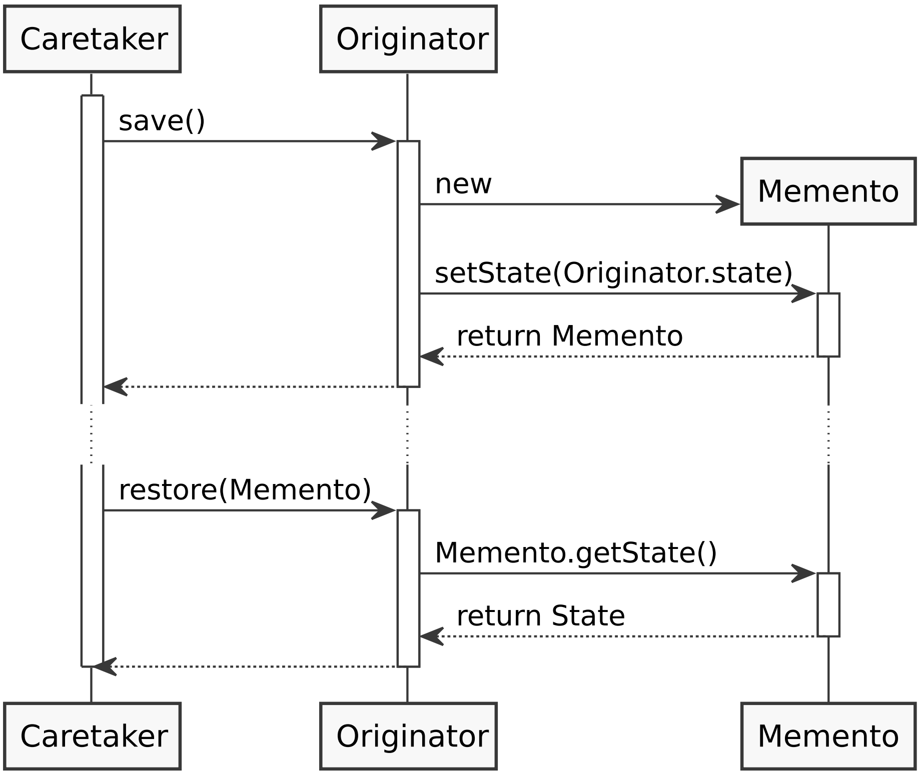
```
public List<TravelPlan> travelTo(fromcity:String, tocity:String,
    from:LocalDateTime, to:LocalDateTime){
      String departure=closestAirport(fromcity);
      String destination=closestAirport(tocity);
      int nights=ChronoUnit.DAYS.between(from, to);
      List<TravelPlan> result=new ArrayList<>();
      for (HotelOffer hotel : hotels.getOffers(from,nights,tocity)){
        //Pick cheapest flights
        FlightOffer fromcity =
            flights.getOffers(departure,destination,from).first();
        FlightOffer tocity =
            flights.getOffers(destination,departure,to).first();
          result.add(new TravelPlan(fromcity, from, flightto, hotel,
              tocity, to, flightback));
      }
      return result;
}
```

c) Usually, users do not immediately book their travel, yet keep several trip plans before they finally book one. As the implementation details of these plans should be hidden from the client, employ the `Memento` design pattern [1]. Draw a corresponding class diagram and implement your design.

Solution: The general structure of the `Memento` design pattern is as follows:

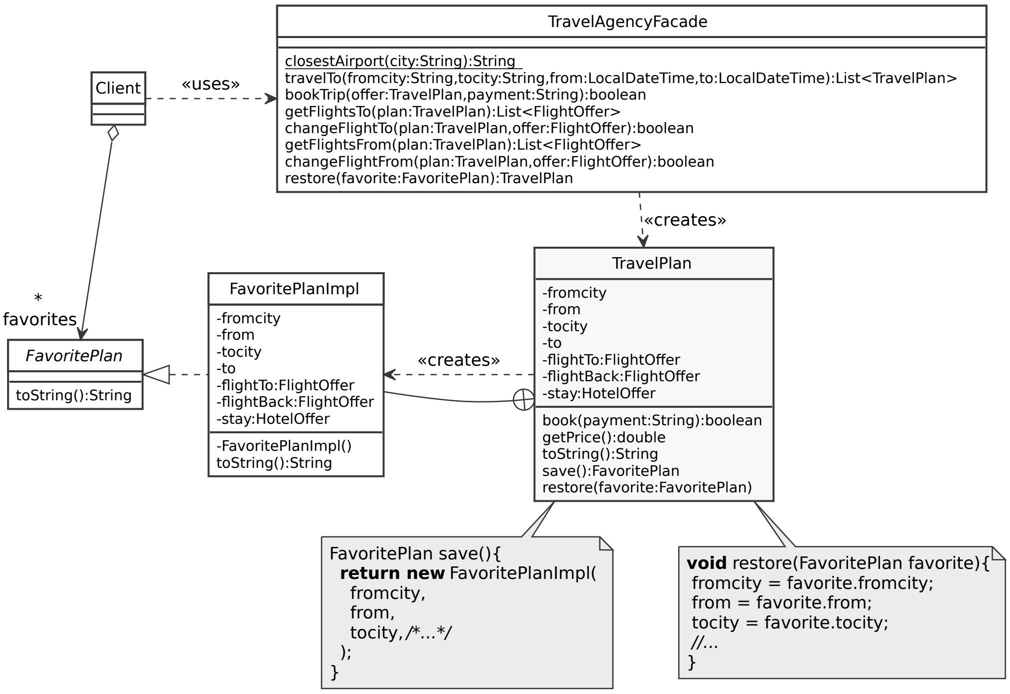


This design permits a `Caretaker` to save the state of the `Originator` and later restore it, without violating the integrity of the internal state of the `Originator`.



Note that the `Originator` can only access the private methods of the `Memento`, if it was implemented as an inner class.

By employing the `Memento` design pattern, the travel agency can be refactored to permit Clients to save their favorite `TravelPlans`, whereas the `FavoritePlan` represents the `Memento`.



Task 3 Homework (optional)

In preparation for the exam, the homework assigns you to explore the relations between the various design patterns.

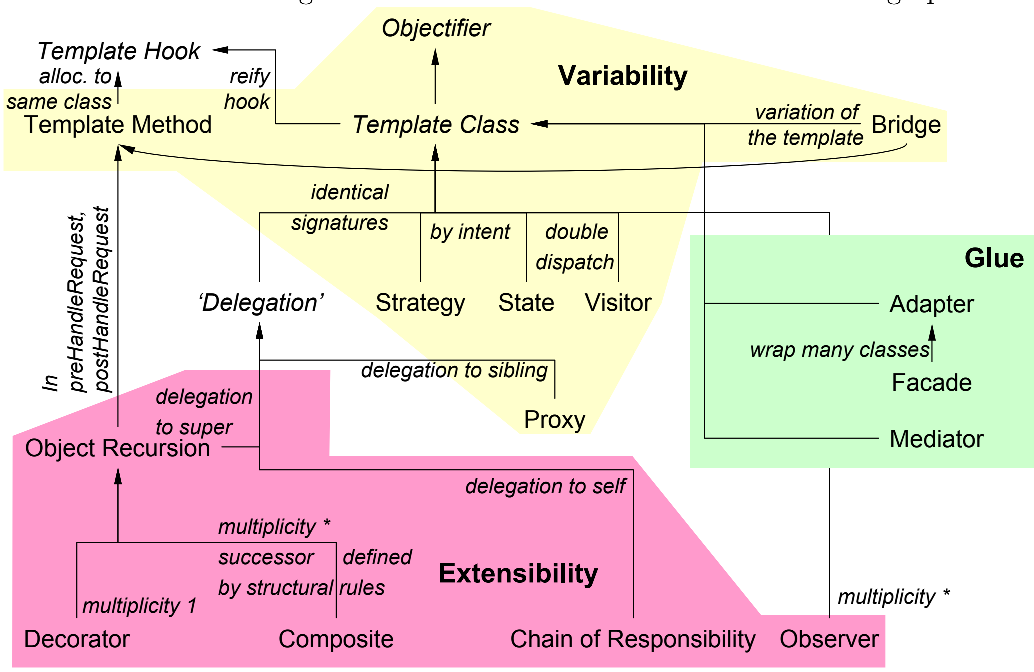
a) Compare `TemplateMethod` and `TemplateClass`. What are commonalities and differences? How do they facilitate variability? What is their relation to the `Template Hook` and the `Objectifier` patterns?

b) Compare the *extensibility* patterns, such as, `Decorator`, `Composite`, `Observer`, and `ChainOfResponsibility`. Which mechanisms permits extensibility? What is the relation of these patterns to `TemplateClass` and `ObjectRecursion`?

c) Now compare the *glue* patterns `Adapter`, `Facade`, and `Mediator`. How do these address architectural mismatch? What is their relation to the *variability* and *extensibility* patterns?

d) Sketch a chart highlighting the relations between the following design patterns `TemplateMethod`, `TemplateClass`, `Objectifier`, `Bridge`, `Strategy`, `State`, `Proxy`, `Visitor`, `Adapter`, `Facade`, `Mediator`, `ObjectRecursion`, `Decorator`, `Composite`, `ChainOfResponsibility`, and `Observer`. Use arrows to indicate specialization, e.g, based on class structure, behaviour, or intent. If necessary, add helper concepts to represent commonalities, which have not yet been abstracted into an individual pattern.

Solution: The following sketch shows the relations of the various design patterns.



References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.