Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

WS2018/19 – Design Patterns and Frameworks

Architecture Mismatch Patterns

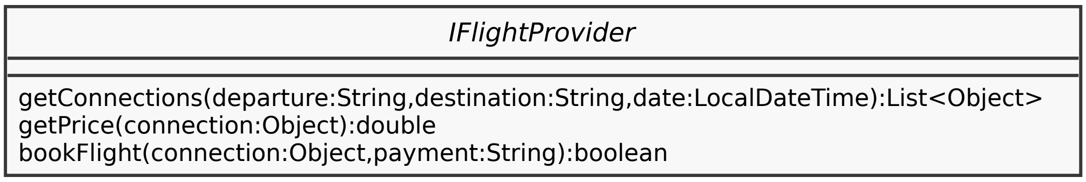
Professor:	Prof. Dr. Uwe Aßmann
Lectuer:	Dr.-Ing. Sebastian Götz
Tutor:	Dr.-Ing. Thomas Kühn

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:

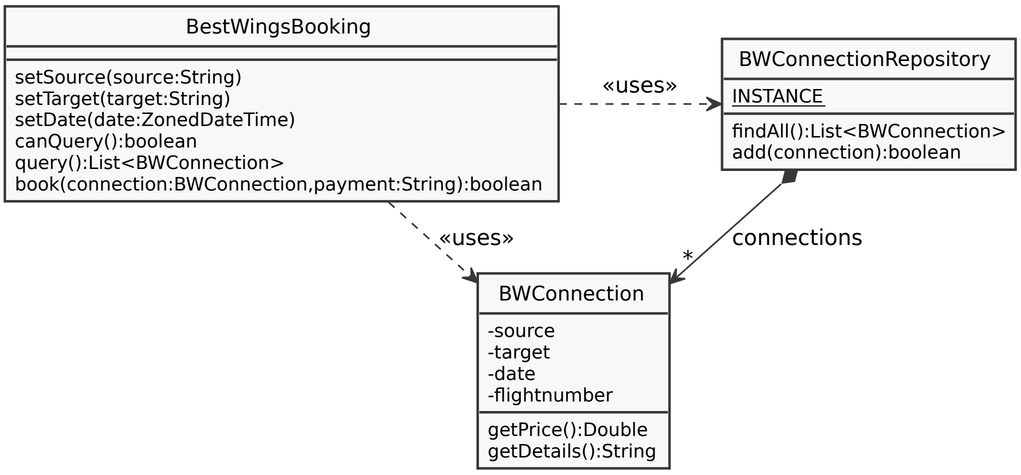
- `getConnections(departure,destination,date):List<Connection>` permits querying all connections from an airport of `departure` to a `destination` airport at a given date (around the given time).
- `getPrice(connection):double` returns the price of the given connection in Euro or throw an exception if 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.



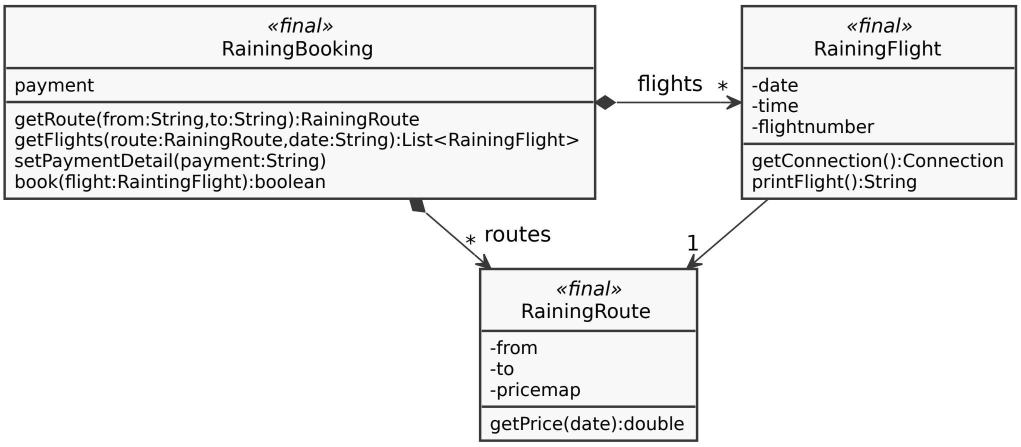
```
IFlightProvider

getConnections(departure:String,destination:String,date:LocalDateTime):List<Object>
getPrice(connection:Object):double
bookFlight(connection:Object,payment:String):boolean
```

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



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


«final»
RainingBooking
payment
getRoute(from:String,to:String):RainingRoute
getFlights(route:RainingRoute,date:String):List<RainingFlight>
setPaymentDetail(payment:String)
book(flight:RaintingFlight):boolean
flights
*
«final»
RainingFlight
-date
-time
-flightnumber
getConnection():Connection
printFlight():String
*
routes
1
«final»
RainingRoute
-from
-to
-pricemap
getPrice(date):double

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.

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?

b) Design and implement the travel booking service.

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. Draw a corresponding class diagram.

Task 3 Homework for Next Exercise

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.

References

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