# SWEN30006 Software Modelling and Design
# Workshop 7: GoF Patterns (Part 1)
# Adapter, Singleton, Factory

School of Computing and Information Systems
University of Melbourne
Semester 2, 2020

## Part 1 Creating Design Model

### Task 1.1 Adapter

Since MiniExpedia will interact with varying interfaces of Search classes in the provided packages, the adapter pattern should be used to wrap the interfaces of those classes. We should also use the adapter for Flight classes since the Search class of each package returns a different class, e.g., the Jetstar Search class returns the JetstarFlight object and the Qantas Search class returns the QantasFlight object. To unify these flight objects, we can use the adapter pattern to wrap these flight classes. Using this pattern, the Search class is not required to know the actual interfaces and classes in the Jetstar, Qantas, and Virgin packages. The Search class only contains and uses the interface classes (i.e., `ISearchAdapter` and `IFlightAdapter`).

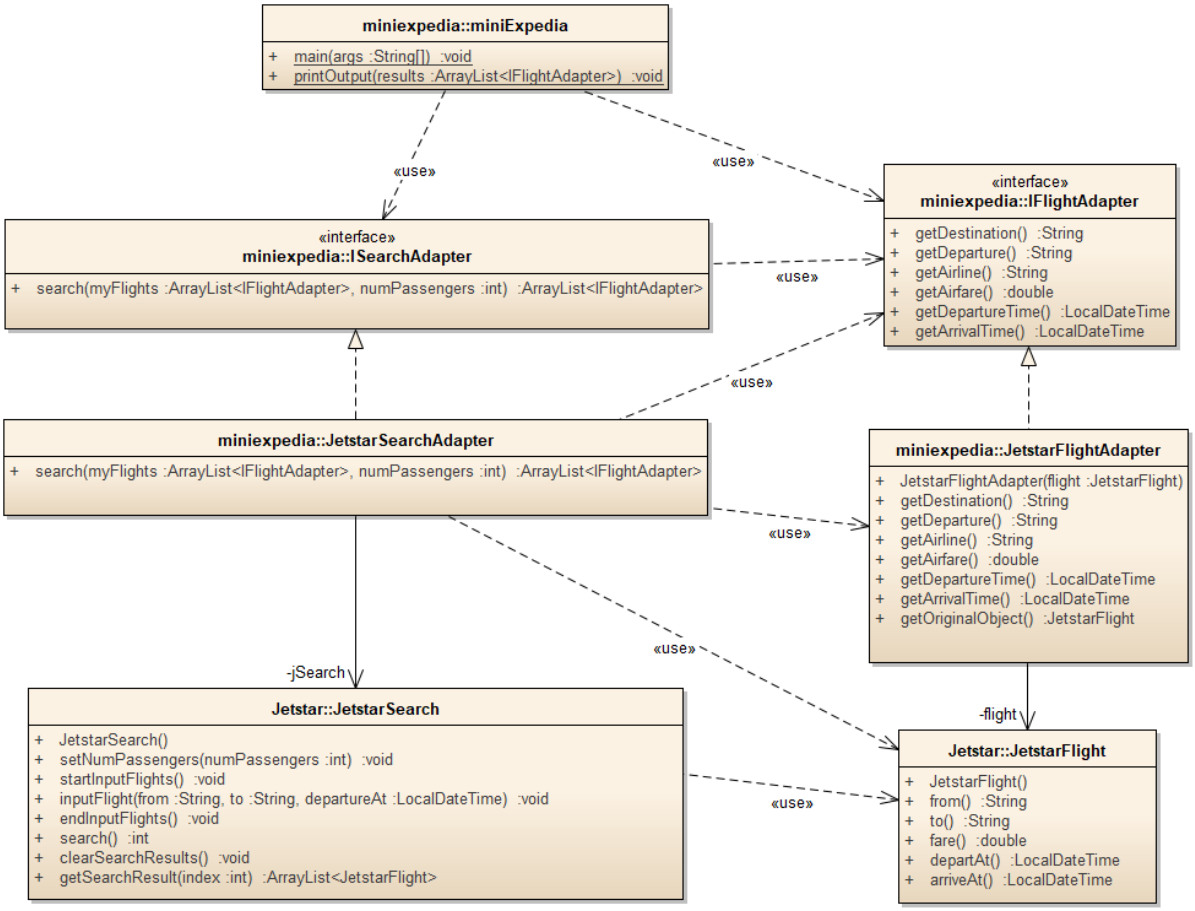
Our aim here is to be able to write code for searching flights which is independent of any particular airline, for example, code of the form shown in the following figure.

```
ISearchAdapter searchAdapter = null;
ArrayList<IFlightAdapter> results = new ArrayList<IFlightAdapter>();
for(int i = 0; i < airlines.length; i++) {
        searchAdapter = ServiceFactory.getInstance().getSearchAdapter(airlines[i]);
        ArrayList<IFlightAdapter> output = searchAdapter.search(myFlights, numPassengers);
        results.addAll(output);
}
```

The diagram below shows an example of implementing adapter for the JetStar Search and Flight classes.

The adapters for the Qantas and Virgin Search and Flight classes should be very similar to the adapter for the Jetstar classes. For example, we can have one new class (`QantasSearchAdapter`) that implements the `ISearchAdapter` class for the Qantas Search adapter and one new class (`QantasFlightAdapter`) that implements the `IFlightAdapter` class for the Qantas Flight adapter. Then, similar to the Jetstar adapters, the `QantasSearchAdapter` links to the Qantas Search class and the `QantasFlightAdapter` links to the Qantas Flight class.

The booking component should be very similar to the Search and Flight components. A new interface class (`IBookAdapter`) should be created. Then, we create a new class that implements this interface class for each airline package. The Book class of each package gets only its flight object, i.e., the Jetstar Book class accepts only the JetstarFlight object based on the function interface, i.e., `requestBooking(flights: ArrayList<JetstarFlight>, ....)`. Therefore, we need the book adapter to unwrap the IFlightAdapter object to its containing object.




If you look the class diagram provided, you will see that we have the `getOriginalObject()` function in the `JetstarFlightAdapter` class. This function should return the `flight` property of this object.

We may not need to have an adapter for the Passenger class in the provided package. This is because we can pass the passenger information (via an object or function arguments) to the Book adapter. Then, the book adapter can create the Passenger object for that particular package, i.e., the JetstarPassenger object can be instantiated in the JetstarBookAdapter class.

## Task 1.2 Singleton Factory

This is exercise is very similar to the example in the Week 7 lecture slides (page 35). Based on the provided user interface in the main class, users can select which airlines will be used. Hence, we can use the singleton factory to determine which adapters should be instantiated.