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

School of Computing and Information Systems
University of Melbourne
Semester 1, 2019

Completion: You need to demonstrate your solutions for **every exercise** to your tutor before the end of the workshop. If you do not complete the exercises in time, finish the remaining tasks by next week's workshop and present these to your tutor at the start of the workshop. Your tutor must review your work for you to be eligible to receive a mark for this workshop. See the Workshop Overview under Subject Information on the LMS for more details.

Requisite Knowledge and Tools

To complete this workshop you will need to be familiar with the following terms and concepts:

- Class Diagram
- Sequence Diagrams
- UML State Machine Diagram
- GRASP principles

Introduction

This week we will be focusing on applying GoF patterns, particularly, Adapter, Factory, and Singleton. The exercises begin with designing a system based on provided information, then analysing which classes should be applied with GoF patterns to solve a design problem. Finally, we will finish by implementing the system and using GoF patterns. These exercises should allow you to gain an understanding of particular design problems through on a concrete example and demonstrate how to apply design patterns to solve the problems. This week's exercises are to be done individually.

GoF Patterns: Adapter, Factory, and Singleton

GoF (Gang-Of-Four) patterns are a set of design patterns that provide solutions to common software design problems. GoF patterns are specialization of the GRASP building blocks. Each GoF pattern is constructed based on a set of principle patterns to address a specific problem. For example, Adapter pattern offers Protected Variations from changing various interfaces through the use of an Indirection object that applies interfaces and Polymorphism. GoF patterns are a useful tool for software developers. These patterns can be considered as generalised solutions or templates that solve real-world problems. However, applying

patterns on a trivial problem may result in overcomplicated implementation. Hence, it is important to understand the concept of the design patterns and their benefits when applying them. Below is a brief summary of three GoF patterns that we will be using in this week's exercises.

- **Adapter:** Adapter pattern is a structural pattern in GoF patterns. It provides a solution on how to link objects that have *incompatible* interfaces. Instead of directly link these objects, Adapter pattern suggests to have an 'adapter', i.e., a class that is responsible for wrapping various interfaces of these objects and providing a single interface. Hence, other objects that want to link with these objects with various interfaces are not required to know all the interfaces, but only one interface provided by the adapter.
- **(Concrete) Factory:** Concrete Factory pattern is a part of Factory Method in GoF patterns. Factory pattern is a creational pattern as it is used to control class instantiation. Creating an object can be a complex task for some problems and may require complex logics to control this creation. To achieve Low Coupling and High Cohesion in GRASP principles, Concrete Factory pattern provides a solution by defining a new class and method that are responsible for generating an object.
- **Singleton:** Singleton pattern is also a creational pattern in GoF patterns. Singleton pattern is useful when we are sure that we need only one instance of a particular class and it will have a global point of access. To address this design problem, Singleton pattern provides a solution by hiding constructor and creating a method that is responsible for instantiate the class itself in order to ensure that the class will not be instantiated from outside the class.

The System Under Discussion: MiniExpedia

This week you will be creating a simple flight searching and booking systems, called MiniExpedia. Like the real-world Expedia and other flight searching services, MiniExpedia provides a service for searching flights from various airlines with specified departure, destination, and departure date. In this workshop, MiniExpedia will retrieve flight information from airlines' services. Below are fundamental steps that the system must have:

- Users can specify information of flights (i.e., departure airport, destination airport, departure date, departure time) and the number of passengers. The searching route can be either **one way, round trip, or multi cities**.
- MiniExpedia sends the search request with specified information to each airline service and present the search results (i.e., departure airport, destination airport, departure date/time of that flight, arrival date/time of that flight, and airfare) from all airlines.
- Users then can select any flight from any airline in the search results for a booking.
- Before a booking, users will specify passenger details, i.e., first name, last name, and date of birth.
- Finally, MiniExpedia sends a booking request with selected flights and passenger details to the corresponding airlines. If the booking request is successful, the airline booking service will provide a confirmation code.

Airline Packages

In this workshop, three packages with javadoc (i.e., Jetstar, Qantas, and Virgin packages) are provided for artificial airline services that you will use to retrieve flights. The class diagrams for these packages are as below.

Jetstar

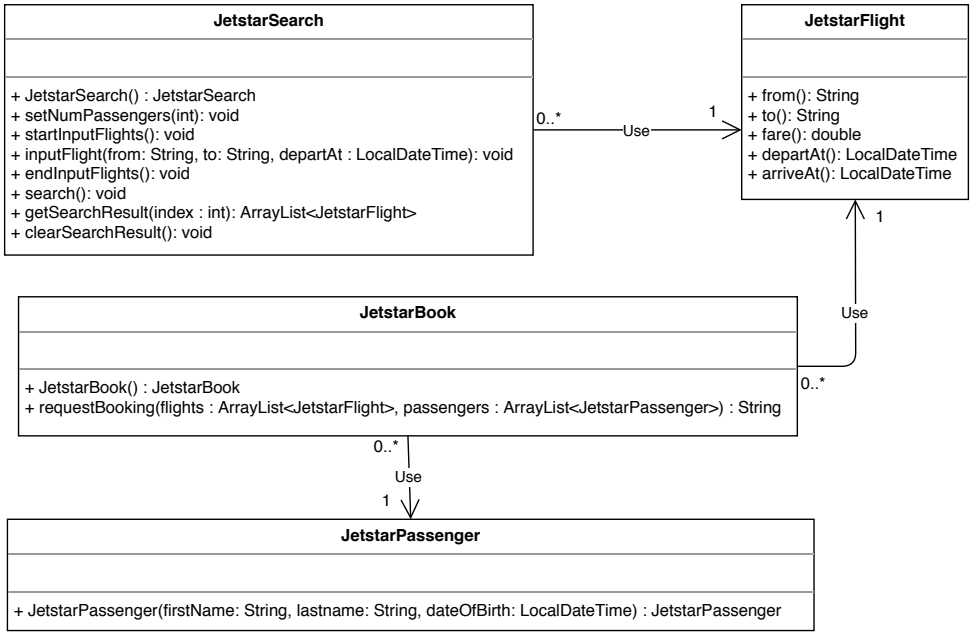

Figure 1: Class Diagram for Jetstar package.

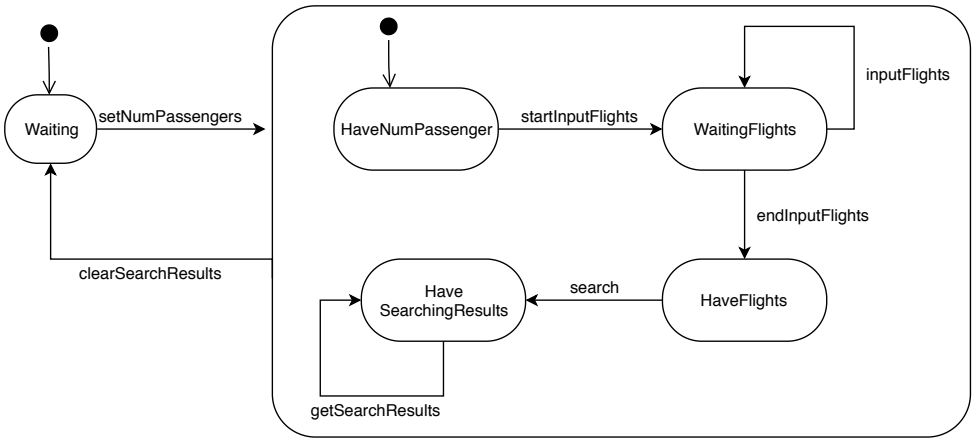

Figure 2: A State Machine Diagram for JetstarSearch class in Jetstar package. **Note:** the Jetstar classes will throw an exception message if the method is called in the invalid state.

Qantas

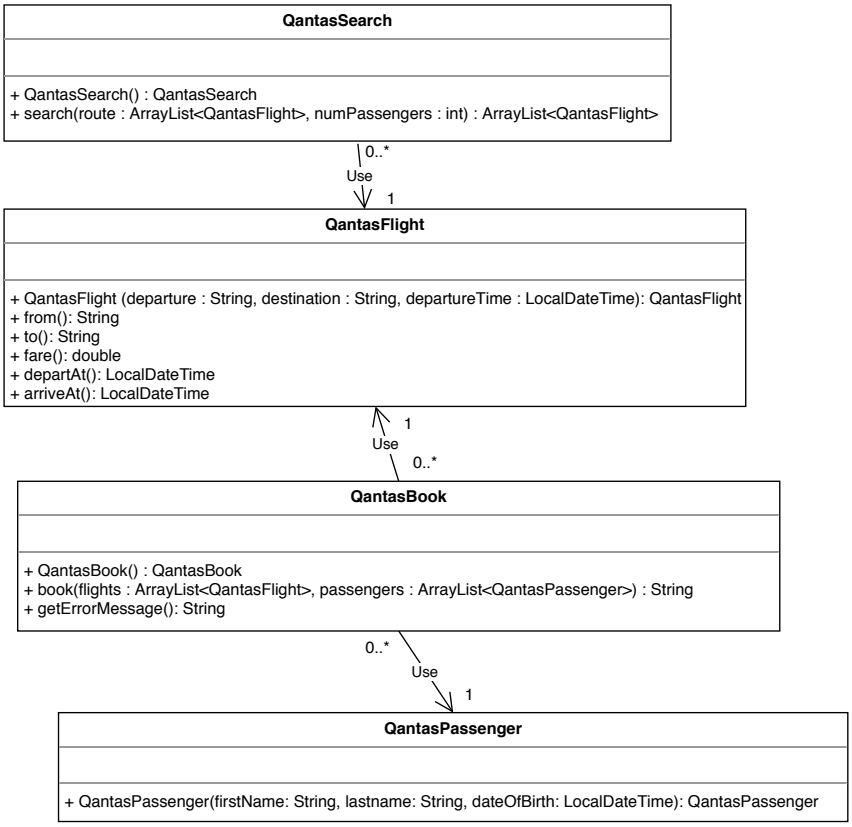

Figure 3: Class Diagram for Qantas package.

Virgin

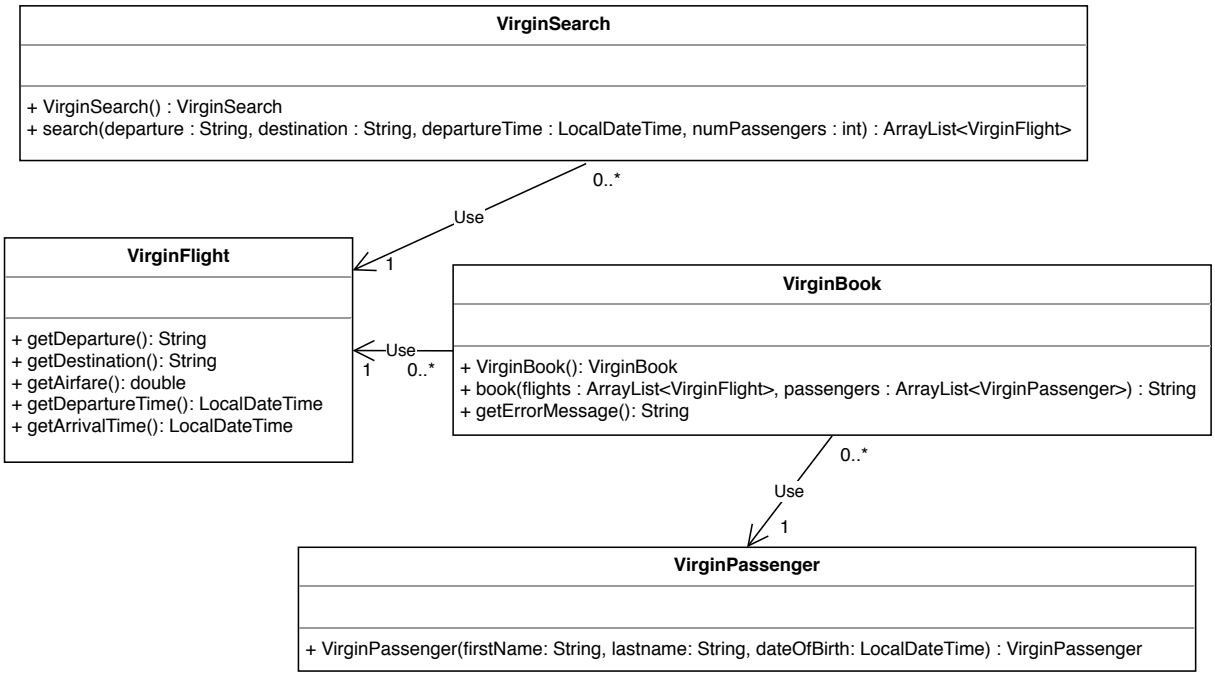

Figure 4: Class Diagram for Virgin package.

Part 1 Designing the System

Before implementing the system, you need to design MiniExpedia in order to understand what data and classes are needed and how will MiniExpedia will interact with the external services (i.e., airline packages).

Task 1.1 Creating Domain Model

Using the use cases described in the previous section regardless the external services (i.e., airline packages), create a **Domain Model** for the problem domain as you understand it.

Task 1.2 Creating Design Model

Based on your Domain Model, you should be able to identify which objects in your Domain Model will link with external services. For example, Search object will link to the airline search objects (i.e., JetstarSearch, QantasSearch, and VirginSearch). As you can notice in the class diagrams in the previous section, those external Search objects have different interfaces and behave differently.

Task 1.2.1 Adapter

Identify all objects that require Adapter. Draw Class Diagrams with Adapter for all corresponding classes. Creating Class Diagrams for Adapter

Task 1.2.2 Singleton Factory

To achieve High Cohesion, the Adapters in Task 1.2.1 should be created by a designated object (i.e., Factory). Draw a System Sequential Diagram to describe how your Adapters for external services will be created and how your objects will send a request to the external services for searching and booking.

Part 2 Implementing the System

Using your design in Part 1, implement your MiniExpedia in Java. You should create the appropriate classes for Adapters and Singleton Factory. An interface to get flight information from users are provided in `MiniExpedia_interface.zip`. It is required to have a consistency between your design and implementation. By now, you should also be aware of GRASP principles and ensure that your design and implementation do not violate GRASP principle.

Instructions for Setting Up Your Project:

1. Download `MiniExpedia_interface.zip` and `AirlinePackages.zip` from the LMS
2. Right click in the Project Explorer (or Package Explorer) window and select **Import...**
3. Select 'Projects from Folder or Archive' (you can use the search bar to find it) and click **Next >**.
4. To the right of 'Import source:' click **Archive...**, then select the `MiniExpedia_interface.zip` file you just downloaded and click Finish.

If **Referenced Libraries** is not shown in the project explorer, do the following steps:

1. Right click on your project. From the menu that pops up select **New**, then select **Folder**
2. Name the new folder **lib**

3. Unzip AirlinePackages.zip
4. Click and drag the three .jar files (**Jetstar.jar**, **Qantas.jar**, and **Virgin.jar**) onto the **lib** folder in your project. When prompted, select to **copy** the files rather than link to them
5. In the Eclipse project explorer, select the three .jar files in the lib folder
6. Right click the .jar files
7. In the menu that pops up, select **Build Path**, then **Add to Build Path**
8. In the project explorer, there should be **Referenced Libraries** added

Minimum Requirement: Two external services are used for searching in your implementation.
