Design Patterns and Frameworks	**Exercise Sheet No. 6**
Dipl.-Inf. Steffen Zschaler	Software Engineering Group
INF 2097	Institute for Software and Multimedia Technology
`http://st.inf.tu-dresden.de/teaching/dpf`	Department of Computer Science
	Technische Universität Dresden
	01062 Dresden

Architecture Mismatch Patterns

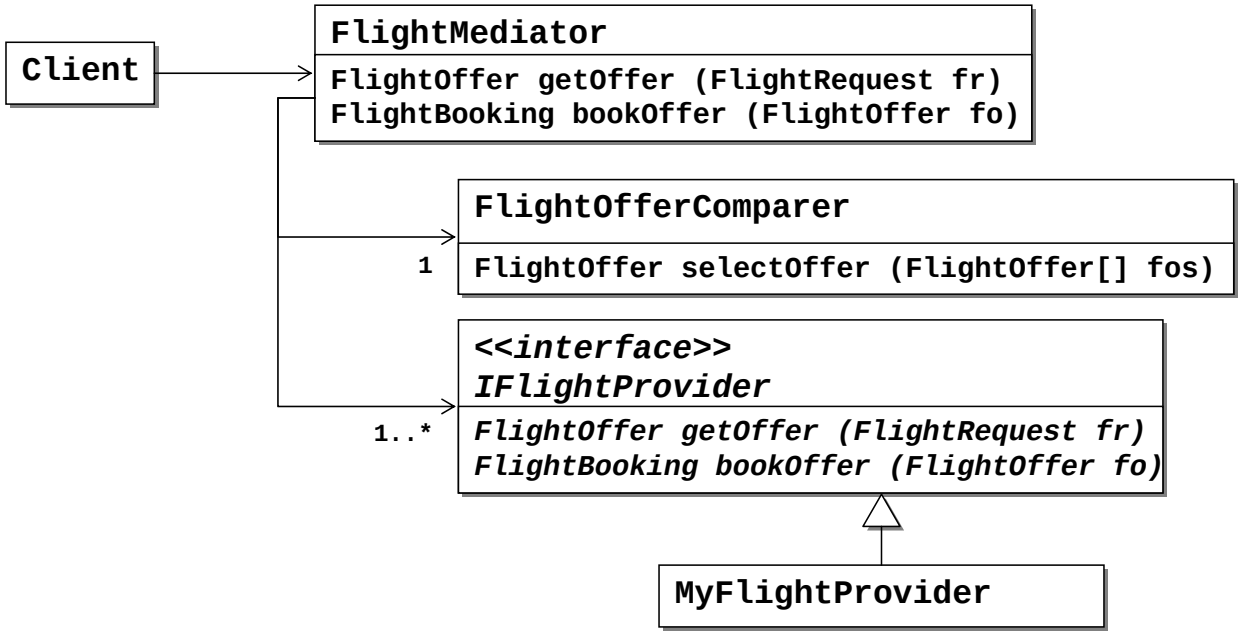
Task 1: Medi(t)ative Air

Design an application which enables you to book the cheapest flight to a destination of your choice out of a number of providers.

1a) **Task:** Assume, every provider is known in advance, and implements an interface `IFlightProvider`, which provides operations for querying for a connection, and for booking a flight. Develop an architecture which enables clients to interface to these providers and book the cheapest flight on offer for the destination and date they are interested in. Flight providers should require (and receive) no knowledge on other flight providers known to the system. Also, clients should not need to know which flight providers are registered with the system.

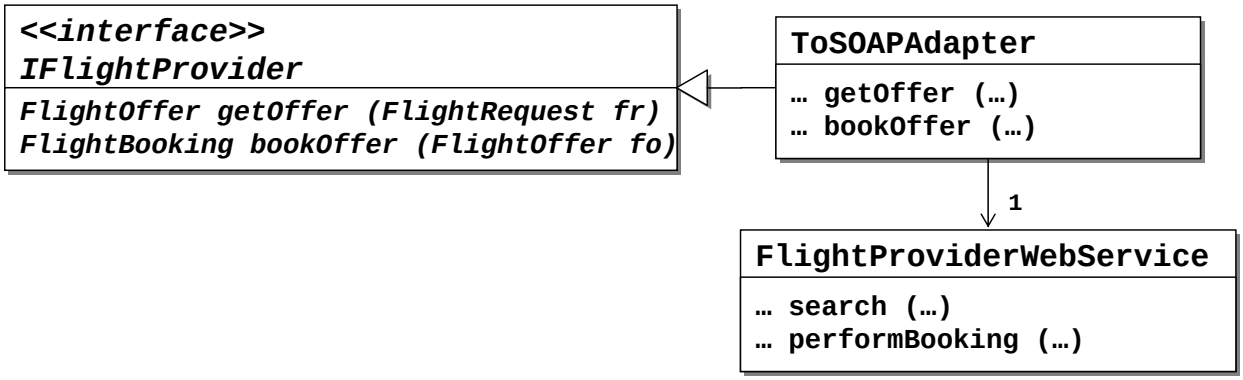
Which design pattern could you use?

Solution: Mediator would be a good design pattern for this. The mediator provides the client's interface and connects the flight providers and the selection component which selects the cheapest flight for booking—of course, after checking back with the client.



1b) **Task:** Many airlines offer on-line booking services as web services. How can you incorporate such an airline as a flight provider?

Solution: The airline can be incorporated by using an ADAPTER which maps the SOAP interface onto `IFlightProvider`. You may need to implement one adapter per airline, because they may use slightly different SOAP protocols.



Task 2: Photo-realistic Facade

Ray tracing is a rather complex technique. It consists of a number of steps from parsing a scene-graph description (often called a 'script'), building a scene-graph instance in memory, optimising the scene graph, tracing rays through all pixels of the target image, possibly oversampling to provide anti-aliasing, to actually rendering the image; that is, transforming the ray colour values into the value range of image colour values. On the other hand, as a client all you want to do is provide a script and obtain an image.

2a) **Task:**

Use the FACADE pattern to provide clients of a ray-tracing subsystem with easy access to ray-tracing functionality.

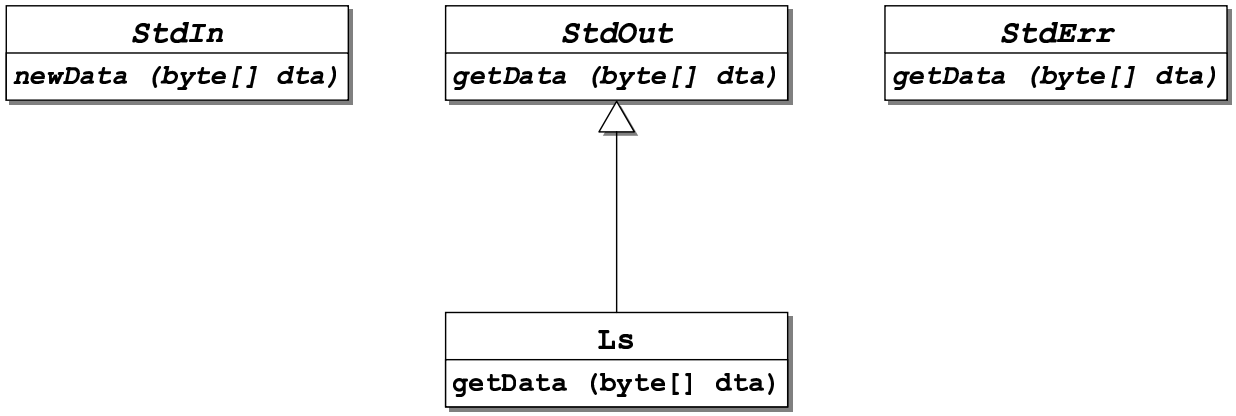
Solution: The facade is a class that uses the subsystem to provide very simple access to a subsystem. A facade for the ray-tracing subsystem may have only one operation: `public Image render (File fScript)`.

Task 3: Producer-Consumer with Mediator

Assume, you are realising a UNIX shell. All programs running on the UNIX shell are provided with three communication channels: `stdin`, `stdout`, and `stderr`.

3a) **Task:** Design interfaces for `stdout`, `stderr`, and `stdin`. Sketch an implementation for these interfaces by the 'ls' command, and how this would be used by your shell implementation.

Solution:



3b) **Task:** The 'argouml' program is a visual editing tool for UML diagrams. Assume that the model being edited is encapsulated completely behind an instance of `ModelFacade`. What design pattern could you use to allow for 'argouml' to be used for manipulating UML models from the command line? It should accept an XMI file on `stdin` and output a transformed XMI file on `stdout`.

Solution: We need to wrap 'argouml' in a 2-WAY ADAPTER to `stdin` and `stdout`.

3c) **Task:** Which design pattern is used by your shell to realise a command like

```
cat mymodel.xmi | argouml - > mytransformedmodel.xmi
```

?

Solution: The shell acts as a simple MEDIATOR.

Task 4: Pattern Relations

In this task you will explore the relations between the various patterns that we have been looking at in the course so far.

4a) **Task:**

Compare TEMPLATE METHOD and TEMPLATE CLASS. What do they have in common, what is the major difference? How do they achieve variability? What is their relation to the TEMPLATE HOOK and the OBJECTIFIER patterns?

Solution: Both patterns achieve variability by separating a fixed template and a variable hook, as described by TEMPLATE HOOK. Their most important difference lies in the allocation of classes for the template and the hook part. TEMPLATE METHOD allocates both operations to the same class, while TEMPLATE CLASS uses a separate class for the hook. TEMPLATE CLASS thus combines OBJECTIFIER and TEMPLATE HOOK.

4b) **Task:**

Compare the extensibility patterns DECORATOR, COMPOSITE, CHAIN OF RESPONSIBILITY, and OBSERVER. What are the mechanisms through which they achieve extensibility? Why does PROXY not provide extensibility? What is the relation of these patterns to TEMPLATE CLASS and OBJECT RECURSION?

Solution: Extensibility is about being able to add an unlimited (and typically not pre-determined) set of objects which can be managed in a uniform manner. This requires

1. either an association with a '*' multiplicity at the target end, which is treated inside `while`-loops,
2. or a recursive reference either to `self` or to a `super` class.

The patterns named pretty much span this field. Proxy cannot provide extensibility because it has neither a '*' multiplicity association nor a recursive reference to `self` or a `super` class. It is therefore strictly a variability pattern.

All extensibility patterns still use Template Class, but they manage multiple (typically a number unknown *à priori*) instances of the hook class. Object Recursion is the basic pattern for recursive associations, which is specialised by both Decorator and Composite.

4c) **Task:**

Now compare the architecture-glue patterns Adapter, Facade, and Mediator. How do they cope with architectural mismatch? How do they compare to the variability and extensibility patterns?

Solution: The architecture-glue patterns are Template Class patterns that perform semantic mappings of interfaces.

4d) **Task:** *

Sketch a chart of the relations between the design patterns Template Method, Template Class, Objectifier, Bridge, Strategy, State, Visitor, Proxy, Adapter, Facade, Mediator, Object Recursion, Decorator, Composite, Chain of Responsibility, and Observer. Use arrows to indicate specialisation (based on class structure, behaviour, or intent) and introduce additional helper concepts if you need them to represent commonalities which have not yet been abstracted into an individual pattern.

Solution: This is my conception of the relations between the patterns: