DPF EXCERCISE #6

TUD, ST Group
Dr. Sebastian Götz

Patterns for Architectural Mismatch

Task #1 – Medi(t)ative Air

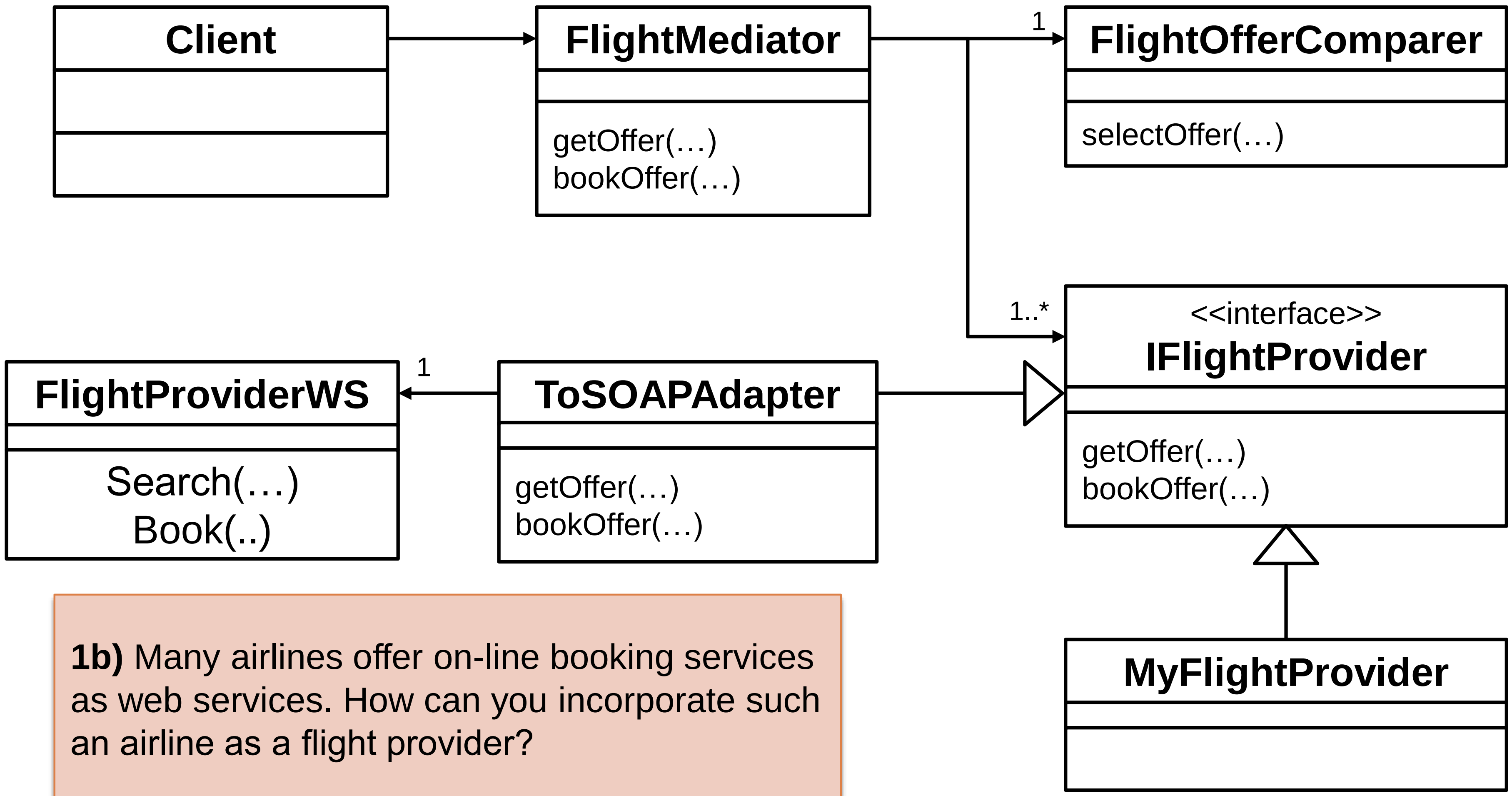
- Design an application which enables you to book the cheapest flight to a destination of your choice out of a number of providers.

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?

Task #1 – Medi(t)ative Air



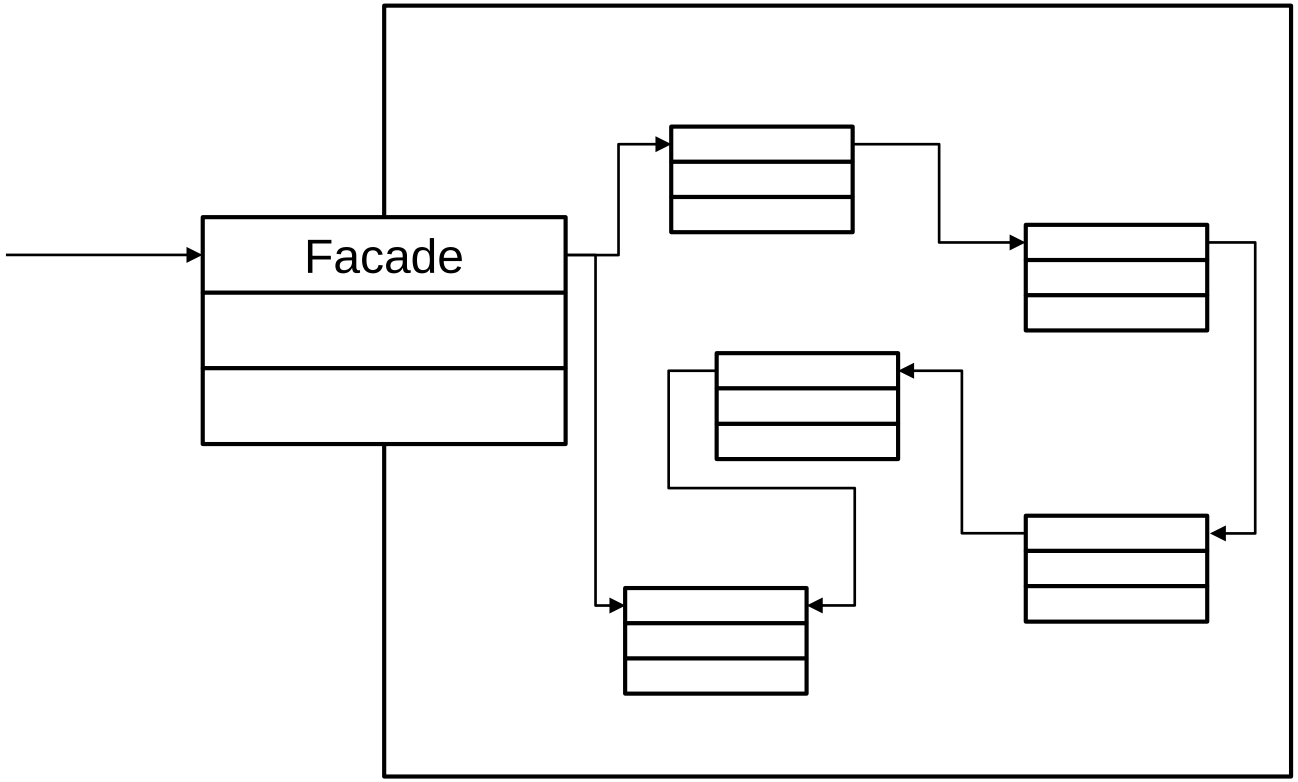
1b) Many airlines offer on-line booking services as web services. How can you incorporate such an airline as a flight provider?

Task #2 – Raytracer

- 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, optimizing 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 color values into the value range of image color values. On the other hand, as a client all you want to do is provide a script and obtain an image.

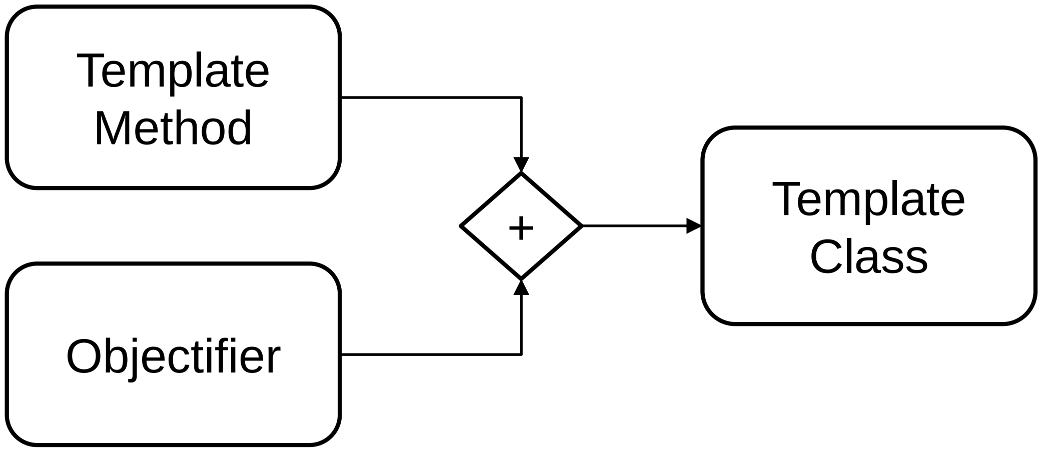
Which design pattern can be used ?

Facade Pattern



Task #3: Pattern Relations

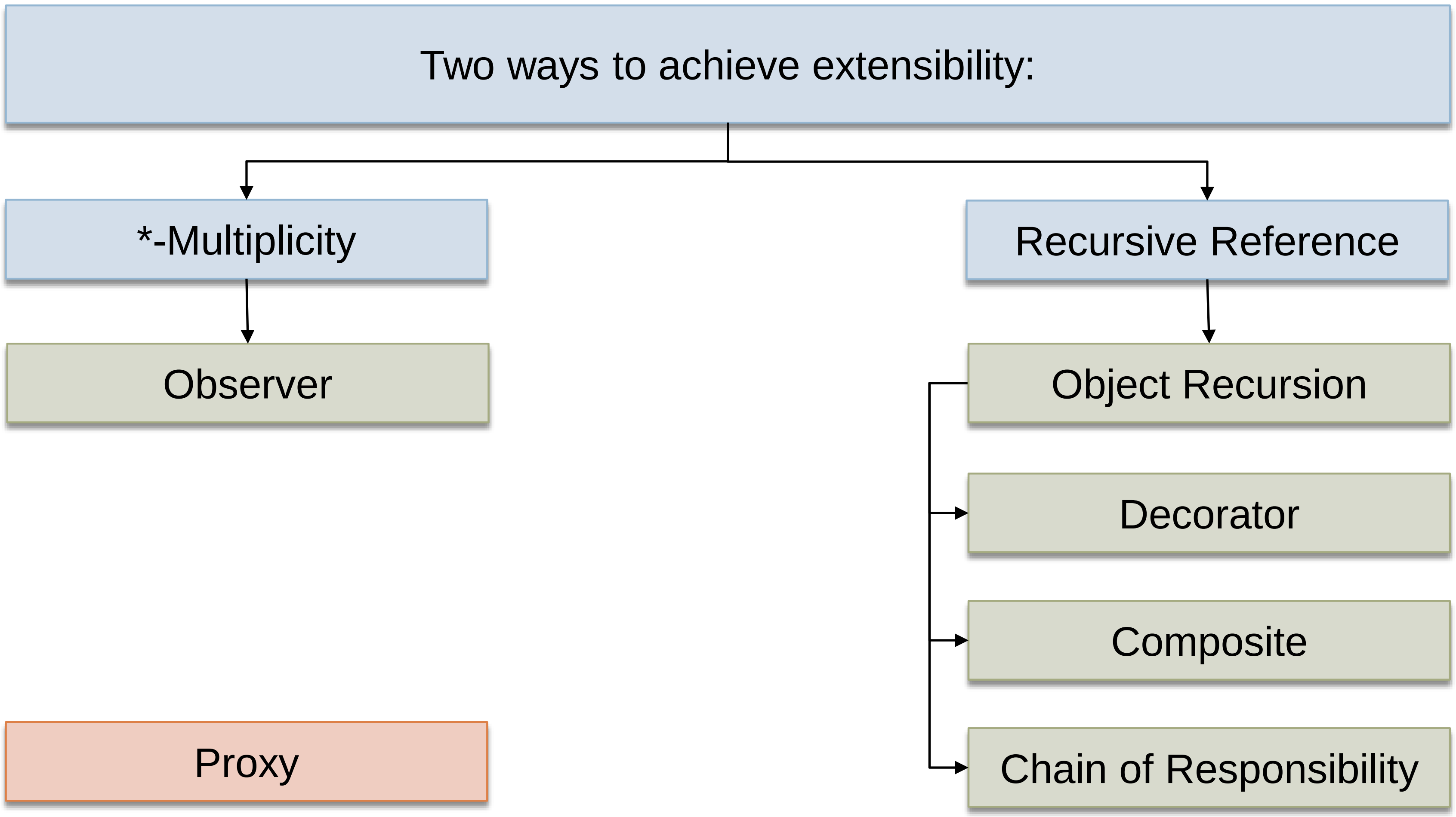
- Compare **Template Method** and **Template Class**. What do they have in common, what is the major difference? How do they achieve variability? What is their relation to the **Template Hook** and the **Objectifier** patterns?



Task #3: Pattern Relations

- 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**?

Task #3: Pattern Relations



Task #3: Pattern Relations

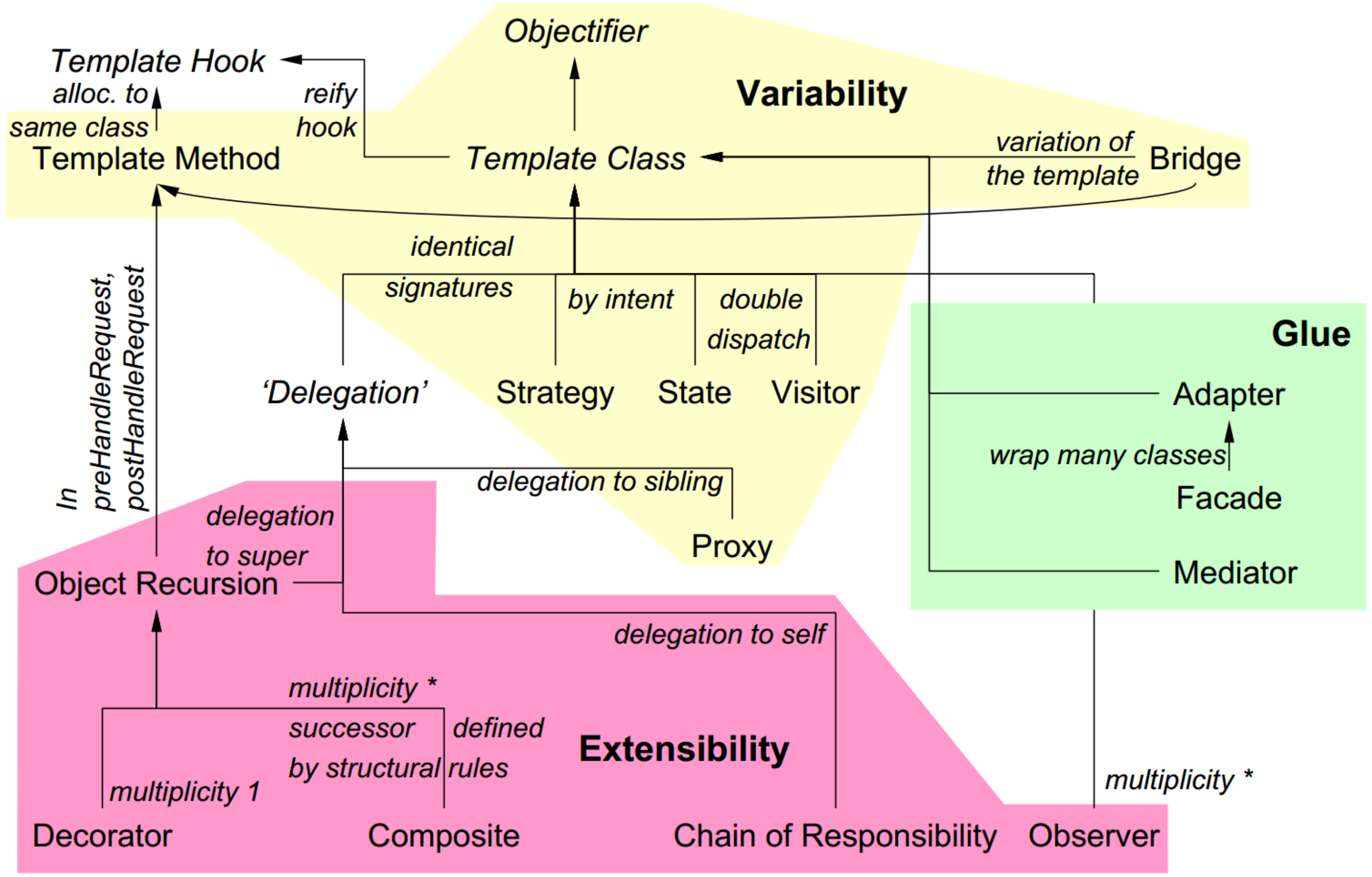
- Now compare the architecture-glue patterns **Adapter**, **Facade**, and **Mediator**. How do they cope with architectural mismatch? How do they compare to the variability and extensibility patterns?

All patterns base on **Template Cass**, but have the intent to perform semantic mappings of interfaces.

Task #3: Pattern Relations

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 specialization (based on class structure, behavior, or intent) and introduce additional helper concepts if you need them to represent commonalities which have not yet been abstracted into an individual pattern.

Task #3: Pattern Relations


Objectifier
Variability
Template Hook
alloc. to same class
reify hook
Template Method
Template Class
variation of the template
Bridge
identical signatures
by intent
double dispatch
'Delegation'
Strategy
State
Visitor
Glue
Adapter
wrap many classes
Facade
Mediator
In preHandleRequest, postHandleRequest
delegation to sibling
Proxy
delegation to super
Object Recursion
delegation to self
multiplicity *
successor by structural rules
defined
multiplicity 1
Extensibility
multiplicity *
Decorator
Composite
Chain of Responsibility
Observer