<http://java.dzone.com/articles/design-patterns-uncovered-1>

FAÇADE PATTERN

**USE:**

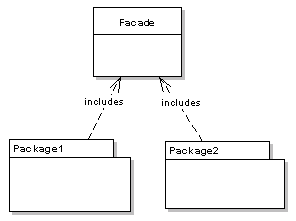
The facade pattern is used when you want to hide an implementation or otherwise make available a different interface externally. The builder/factory pattern is used when you want to hide the details on constructing instances.

**The Facade Pattern**

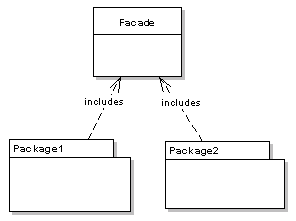
Like the Adapter pattern, Facade is known as a **structural**pattern, as it's used to identifying a simple way to realize relationships between entities. The definition of Facade provided in the original Gang of

Four book on Design Patterns states: ***http://www.dofactory.com/Images/pixel.gifProvide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.***

The diagram definition of the Facade pattern is quite simple - all you're really doing is insulate client from the subsystem:

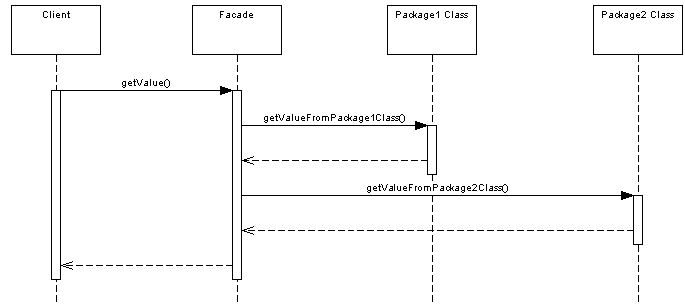


The diagram definition of the Facade pattern is quite simple - all you're really doing is insulating client from the subsystem:



**Like the adapter pattern, the Facade can be used to hide the inner workings of** a third party library, or some legacy code.  All that the client needs to do is interact with the Facade, and not the subsystem that it is encompassing.

The following sequence diagram illustrates how the pattern is used by a client:



### Where Would I Use This Pattern?

As the concept behind facade is to simplify an interface**, service oriented architectures make use of the facade pattern.**

**For example,** in web services, one web service might provide access to a number of smaller services that have been hidden from the caller by the facade. Similarly, a typical pattern in OSGi bundles is to provide an interface package that is exposed to users of the bundle. All other packages are hidden from the user.

### So How Does It Work In Java?

Let's put together a simple example in Java code to illustrate the pattern. Let's take a travel agent site for example, that allows you to book hotels and flights. We have a HotelBooker:

[view source](http://java.dzone.com/articles/design-patterns-uncovered-1#viewSource)

[print](http://java.dzone.com/articles/design-patterns-uncovered-1#printSource)[?](http://java.dzone.com/articles/design-patterns-uncovered-1#about)

01.public class HotelBooker

02.{

03.

04.public ArrayList<Hotel> getHotelNamesFor(Date from, Date to)

05.{

06.//returns hotels available in the particular date range

07.

08.}

09.

10.}

And a FlightBooker:

[view source](http://java.dzone.com/articles/design-patterns-uncovered-1#viewSource)

[print](http://java.dzone.com/articles/design-patterns-uncovered-1#printSource)[?](http://java.dzone.com/articles/design-patterns-uncovered-1#about)

01.public class FlightBooker

02.{

03.

04.public ArrayList<Flight> getFlightsFor(Date from, Date to)

05.{

06.//returns flights available in the particular date range

07.

08.}

09.

10.}

Both of these have Hotel and Flight datatypes, which the client has knowledge about. They could be provided in the same package as the Facade for example.

The TravelFacade class allows the user to get their Hotel and Flight information in one call:

[view source](http://java.dzone.com/articles/design-patterns-uncovered-1#viewSource)

[print](http://java.dzone.com/articles/design-patterns-uncovered-1#printSource)[?](http://java.dzone.com/articles/design-patterns-uncovered-1#about)

01.public class TravelFacade

02.{

03.

04.private HotelBooker hotelBooker;

05.private FlightBooker flightBooker;

06.

07.public void getFlightsAndHotels(Date from, Data to)

08.{

09.ArrayList<Flight> flights = flightBooker.getFlightsFor(from, to);

10.ArrayList<Hotel> hotels = hotelBooker.getHotelsFor(from, to);

11.

12.//process and return

13.

14.}

15.

16.}

All that the client needs to worry about is the Facade class:

[view source](http://java.dzone.com/articles/design-patterns-uncovered-1#viewSource)

[print](http://java.dzone.com/articles/design-patterns-uncovered-1#printSource)[?](http://java.dzone.com/articles/design-patterns-uncovered-1#about)

01.public class Client

02.{

03.

04.public static void main(String[] args)

05.{

06.TravelFacade facade = new TravelFacade();

07.facade.getFlightsAndHotels(from, to);

08.}

09.}

As you can see, it's just a simple approach to encapsulating data.

difference between facade and factory pattern

**The facade pattern is used when you want to hide an implementation** or it is about changing interface of some class or set of classes. Builder hides the process of construction by decomposing it in smaller steps.

**Abstarct factory pattern is used when you want to hide the details on constructing instances.**  
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**EXAMPLE IN JDK:**

1. Facade: [ExternalContext](http://docs.oracle.com/javaee/6/api/javax/faces/context/ExternalContext.html) behaves as a facade for performing cookie, session scope and similar operations.