1.1 Service Façade Pattern

1.1.1 Name and Source

Service Facade
Page 171-178 in the book ".NET Patterns: Architecture, Design, and Process " [NET 03]

1.1.2 Also Known As

-

1.1.3 Type

1.1.4 Intent

To provide a web service controller class as an entry point to a business logic façade object, which can be reused from other contexts than web services, e.g. from within a CORBA based application.

1.1.5 Problem

Some platforms have certain framework classes that your web service implementation is supposed to inherit from. For example, if a web service is created with .NET the baseclass is often “System.Web.Services.WebService” even though you do not have to inherit from it if you do not need to access the common ASP.NET objects.

If your framework does not support multiple inheritance, you will then not be able to inherit from another class you might want to inherit from.

A web service implementation class that uses SOAP code may throw SOAP related exceptions, which you would not want to catch from an application where another communication protocol is used, for example CORBA.

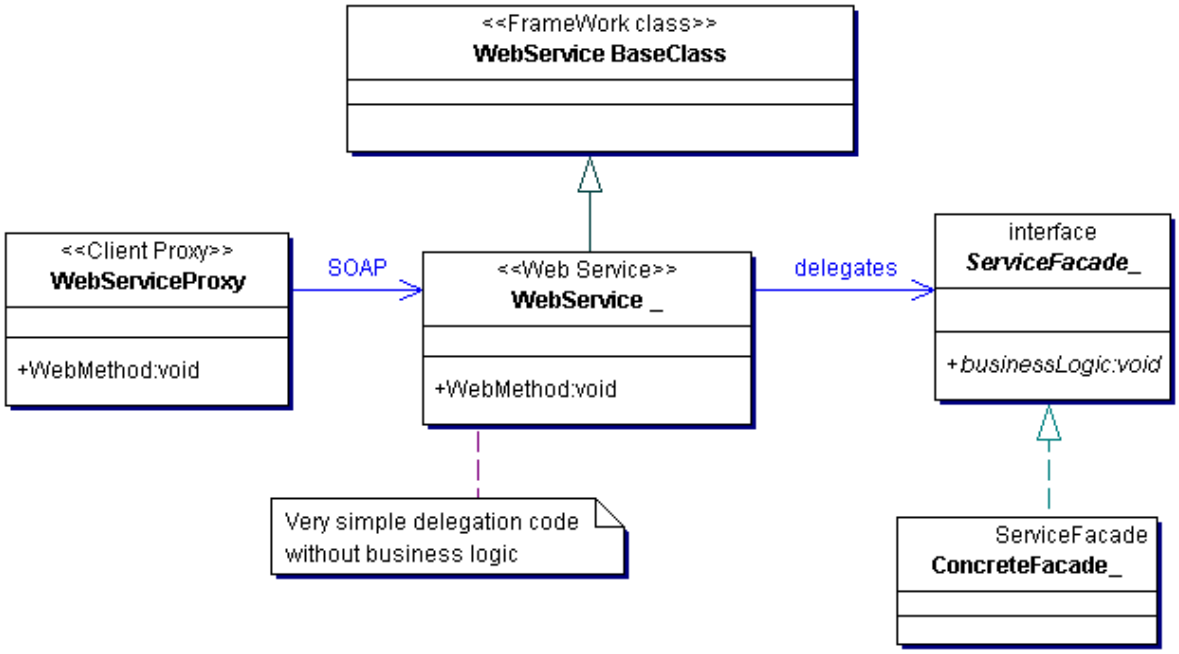
How can you reuse the business logic used by a web service from another context such as CORBA, without risking to catch a SOAP exception, while also being able to inherit from any class you want without wasting your only inheritance possibility by inheriting from a framework class ?

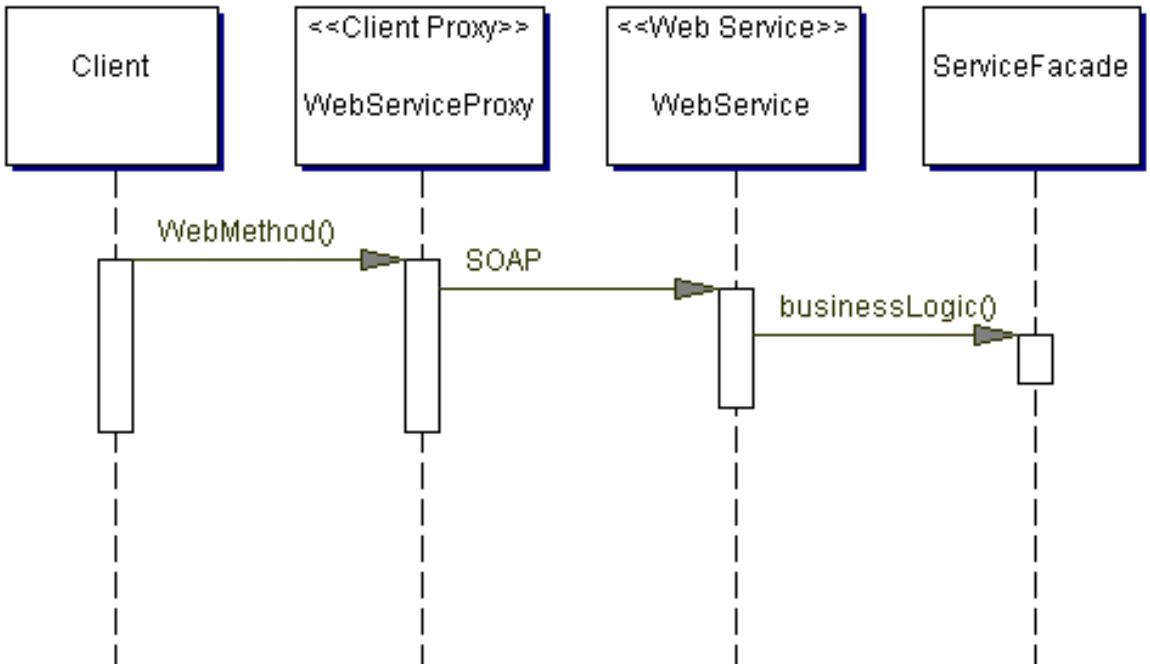
1.1.6 Forces

1.1.7 Solution

Put all the business logic in a service façade class, and use very simple code in the implementation of the web service that essentially only delegates the invocations to the service façade and tries to catch possible application exceptions that then will be rethrown as SOAP

exceptions.





WebServiceProxy – A client sided proxy object that communicates with the web service using the SOAP protocol.

WebService – An implementation of a web service that may want to, or have to, inherit from a framework provided web service base class. It simply delegates invocations to the ServiceFacade, and it also may want to catch application specific exceptions from the ServiceFacade and rethrow them as SOAP exceptions.

ConcreteFacade – An object that contains the business logic. It can implement a ServiceFaçade interface but may also inherit implementation from some baseclass, which would not be possible (in a single inheritance language) if this class would be a web service itself within a framework that enforces inheritance from a framework class. This class should not throw any protocol related exception, such as a SOAP exception, since it may be used not only from the WebService object but also from another context such as a CORBA application that should not be forced to handle any SOAP exceptions. You can use a ConcreteFacade directly and it is not necessary to use any ServiceFacade interface as in the diagram, but if such an interface is used then the pattern can be combined with a "Unchained Service Factory" (or "Chained") as a generic web service controller that, by parsing XML metadata, chooses which concrete façade to use for invoking the business logic.

1.1.8 Consequences

If this "Service Façade Pattern" is used from the "Chained Service Factory Pattern" (or "Unchained SFP") i.e. if the façade object is used to parse XML from the web service requets then you can let the façade create a "Context Object" [CJP 03] from the XML and then add an extra layer that receives a context object, which also could be constructed without depending on XML, i.e. be reusable from other contexts such as CORBA.

The "consequences" section in the source book seems to be contradictional with the description of the pattern. One of the subsections has the title "*Provides an entity in which to house Web Service-specific parameter passing*". Note that the sentence simply starts with "Provides" and not with "The Façade provides" or "The Web Service class provides" so it is unclear what it refers to. In my opinion, to make the pattern description consistent it should refer to the "Web Service" class which should take care of anything that is specific to the Web Service and then delegate to the Façade. However, the subsection later says that the web service class itself should not be cluttered with this type of logic.

Another confusing thing is the description of the ConcreteFacade in the participants section, when it says that "*Most business rules are contained or driven from here, especially those related to packaging data that must be sent back to the Web Service client*". According to that description it seems to me as if the façade object assumes a Web Service client and thus may not easily be reused from a context that not easily can understand that "packaging" of data.

Since I now have made two quotes above that in my opinion are confusing and contradict my general description of the pattern, I would also like to quote at least one statement from the book that supports my overall comprehension of the pattern: "*the Service Façade must take into account the possibility of both Web methods acting as a client to the Façade and other non-Web-Service clients*".

1.1.9 Related patterns

The pattern can be combined with the "Unchained Service Factory" (or "Chained") as a generic web service controller that, by parsing XML metadata, chooses which concrete service façade to use for the business logic. That concrete façade may then also parse the XML to determine the details about the invocations, as described in the Chained/Unchained Service Factory pattern. However, that parsing could also be put into the web service implementation since it may be preferred to let the ServiceFacade provide a strongly typed interface rather than forcing other kind of applications such as a CORBA application to invoke the façade objects with XML data as parameter.

The J2EE pattern "Business Delegate" [CJP 03] also discusses the concept of throwing application specific exceptions rather than distributing service level exceptions to clients, the same way that the "ConcreteFacade" class in this pattern should not throw any SOAP exception.

The “Remote Façade” [PEAA 02] is a general distribution pattern that provides a coarse-grained façade on fine-grained objects, and just like the web service class in this “Service Façade pattern” it does not contain any business logic but only delegates the method invocations.

The “Service Interface” [ESP 03] is a similar pattern that describes how to provide a web service interface to application logic that can be reused from different contexts, for example through .NET remoting. Actually, the web service class that acts as an entry point in this “Service Façade pattern” can be considered as the “Service Interface” that delegates to the application logic in the “Service Façade” class. Note that the “Service Façade” class may contain self-cotained business rules without any delegation to business objects, according to the description in [NET 03]. Therefore I do not think that “Façade” is a good name, because I agree with Martin Fowler in [PEAA 02] where he at page 391 argues that the “Session Façade” [CJP 03] should not be called a façade since it contains business logic.

The “Web Service Broker” [CJP 03] is also a pattern that describes how to take care of the XML in a SOAP invocation and then delegate to the application logic that may be reused from other non web service contexts. The same thing is also illustrated in the "Delegate Adapter Strategy" in the "Business Delegate pattern" [CJP 03] where an object called "B2Badapter" takes care of the conversion to and from XML and delegates to the business logic handled by the BusinessDelegate object.

Finally, this pattern may also be considered as common sense by people that are used to implement applications with multiple layers, i.e. to use the “Layers Pattern” [POSA 96], and you may think it is obvious that you should not mix business logic within a big web service implementation class, with low cohesion, that also handles the protocol level communication. In other words, you should want to follow the general design objective about separation of concerns. Do not use any protocol-specific information (i.e. SOAP related classes/interfaces) from the classes that you want to be able to reuse from different contexts, but instead use the “Context Object Pattern” [CJP 03] to wrap such information.