1.1 Architecture Adapter Pattern

1.1.1 Name and Source

Architecture Adapter Pattern
Page 57-74 in the book "Web Service Patterns: Java Edition" [WSP 03]
http://www.perfectxml.com/WebSvcPatterns.asp

1.1.2 Also Known As

-

1.1.3 Type

Architectural design pattern.

1.1.4 Intent

To enable easy communication between different architectures as if they were one single component and to do this without forcing the application programmers to first manually create classes, which deal with the low-level and protocol-specific programming.

1.1.5 Problem

By using Web Services you can let different non-compatible architectures (e.g. applications written in different programming languages) communicate with each other by letting the client send XML-code to a server as a SOAP message. These kind of messages may include information about which method that should be invoked and what values that the method parameters will have. In a similar way, the server can send back the result to the client as a SOAP message. The XML data included in these SOAP messages must be parsed in both sides (client and server) and be translated to regular method calls and return values in the native programming languages at the client and server respectively.

How can you avoid the repetitive task of doing similar low-level XML-parsing or SOAP programming each time you will implement or consume a new Web Service ?

1.1.6 Forces

Parsing and programmatically creating XML documents can be a time-consuming task, which is repetitive and needs to be done every time a Web Service is created. Many programming languages have frameworks that let you avoid the lowest level of XML-programming (DOM or SAX-parsing) and can provide SOAP-classes for you, e.g. the Apache Axis (java) framework include classes like “SOAPEnvelope”, “SOAPHeader”, “SOAPBody” and “SOAPFault”.

It would be even better if you also could avoid that kind of SOAP-specific programming and to be able to focus entirely at the business objects. In other words, it would be better to let all the XML and SOAP related code be generated automatically.

1.1.7 Solution

You can use a code-generating tool, which can take a WSDL-file as input and then generate a proxy class that will take care of the XML and SOAP code. Such a tool is included in e.g. the Apache Axis (java) framework, and by using it the application programmer can avoid doing any low level XML or SOAP programming but can instead focus at the business methods. This solution always applies to the Web Service client programmer but may also be used by the Web

Service provider by letting the tool and the WSDL interface generate empty server skeleton classes, which the programmer then will implement to delegate invocations to business methods. However, at the server side you may instead choose to do the generation the other way round, i.e. the server interface can be defined manually in the programming language used at the server, and a tool can then from that interface generate the WSDL file to be used by the clients and possible also by other service providers that will want to generate skeleton classes that implement the same service interfaces.

The UML "class diagram" and sequence diagrams below illustrate the pattern. Note that the so-called class diagram is not a quite real class diagram but should rather be considered as a conceptual diagram that looks a lot like a UML class diagram. More specifically, if the server is programmed in java for example, there will **not** necessarily be, as opposed to what the diagram below indicates, any java code like this: "public class ArchitectureB_Adapter implements CommonArchitectureInterface". The purpose of the interface component in the diagram below is to represent the SOAP interface that the different architectures have agreed upon to use for the communication. As you can see below, the names of the methods in the classes which implement the web service (adapter B and C) does not have to be equal to the name of the SOAP XML method as defined in the WSDL file, which it of course would have to be if we were dealing with a real native programming interface. In the real world, the client will typically not send the SOAP message to any particular application specific interface as in the diagram, but rather the message will be received by a SOAP engine which then will delegate it to the appropriate class, for example to the class "ArchitectureB_Adapter" in the diagram.

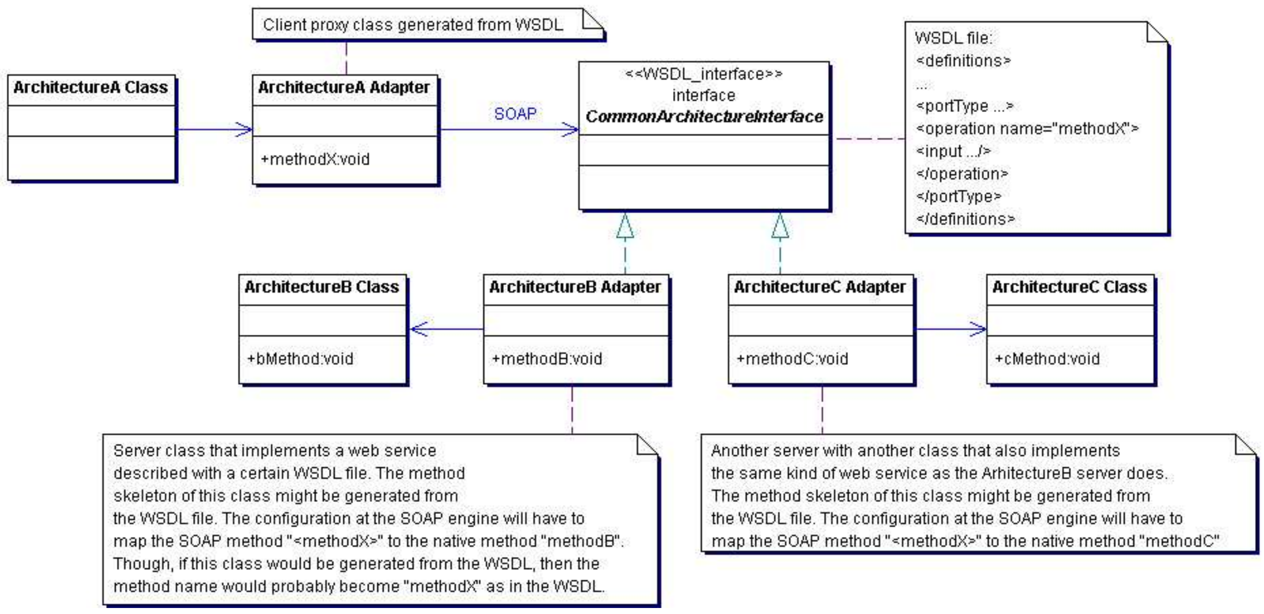

Client proxy class generated from WSDL
ArchitectureA Class
ArchitectureA Adapter
+methodX:void
SOAP
<<WSDL_interface>>
interface
CommonArchitectureInterface
WSDL file:
<definitions>
...
<portType ...>
<operation name="methodX">
<input .../>
</operation>
</portType>
</definitions>
ArchitectureB Class
+bMethod:void
ArchitectureB Adapter
+methodB:void
ArchitectureC Adapter
+methodC:void
ArchitectureC Class
+cMethod:void
Server class that implements a web service described with a certain WSDL file. The method skeleton of this class might be generated from the WSDL file. The configuration at the SOAP engine will have to map the SOAP method "<methodX>" to the native method "methodB". Though, if this class would be generated from the WSDL, then the method name would probably become "methodX" as in the WSDL.
Another server with another class that also implements the same kind of web service as the ArhitectureB server does. The method skeleton of this class might be generated from the WSDL file. The configuration at the SOAP engine will have to map the SOAP method "<methodX>" to the native method "methodC"

Figure 1. Conceptual diagram for the Architectural Adapter design pattern, with UML class diagram notation.

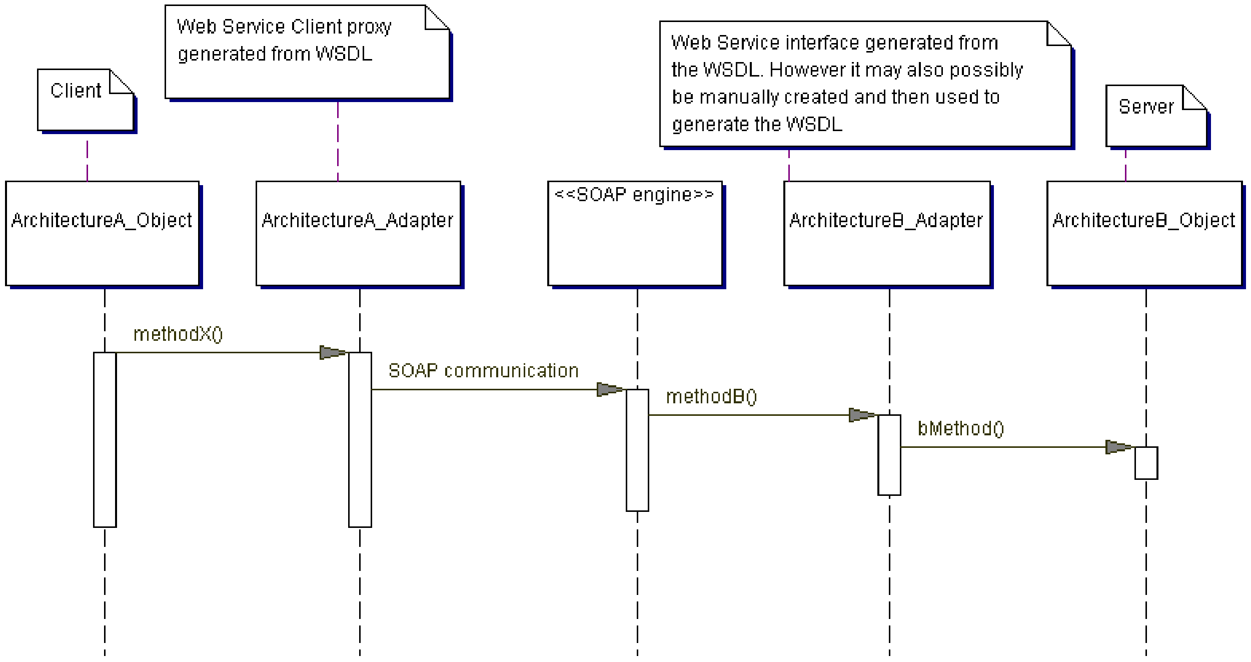

Client
Web Service Client proxy generated from WSDL
Web Service interface generated from the WSDL. However it may also possibly be manually created and then used to generate the WSDL
Server
ArchitectureA_Object
ArchitectureA_Adapter
<<SOAP engine>>
ArchitectureB_Adapter
ArchitectureB_Object
methodX()
SOAP communication
methodB()
bMethod()

Figure 2. Sequence diagram for the Architectural Adapter design pattern.

- ArchitectureB_Object – A program that wants to be available through a Web Service interface, for example an instance of a C++ class, but it does not necessarily have to be a class in an object-oriented language.
- ArchitectureB_Adapter – A component, e.g. a C++ class that may be automatically generated from a WSDL file. If it is generated from WSDL it may have empty methods that the service provider programmer implements by delegating invocations to the business methods in ArchitectureB_Object. This component is written or generated in the same programming language as ArchitectureB_Object while it communicates with clients that may be written in other programming languages, e.g. ArchitectureA_Adapter, with SOAP messages (indirectly through the SOAP engine). If this class is manually created it can be used for generating the WSDL file, which then can be used by clients and other service providers that wants to provide a web service with the same interface, e.g. the server skeleton for the ArchitectureC_Adapter class in the conceptual diagram above the sequence diagram.
- SOAP engine – A Web Service engine that will delegate incoming SOAP messages as method calls to the right web service. Which method to invoke can be determined in different ways by a SOAP engine, e.g. by using the SOAPAction http header, or by looking at the namespace URI:s in the SOAP envelope. The programmer can also implement XML parsing to extract metadata that defines which method to invoke, as described in the "Chained Service Factory pattern".
- ArchitectureA_Adapter – A component, e.g. a java class, that will be automatically generated from a WSDL file, and the generated proxy class will delegate method invocations from ArchitectureA_Object to ArchitectureB_Adapter by sending SOAP messages to the SOAP engine.
- ArchitectureA_Object – A program that wants to be a Web Service client, e.g. a java class, though it does not necessarily have to be a class in an object oriented language.

1.1.8 Consequences

The solution relies on the existence of a code-generating tool, and therefore it may not be generally applied to every programming language (unless you maybe are willing to create the tool yourself for your target language). The solution can also make your application dependent of the code generating tool and you may therefore get problem if your Web Services business partners will start using more recent SOAP and WSDL specifications than your tool currently supports. If you are developing the tool yourself it may be costly to keep up with the last specifications. When you rely on code generation you will also lose some control, e.g. if you are using SOAP headers your tool might not be able to help you to generate that kind of code.

By encapsulating the network code within an architecture adapter object, you can for testing reasons replace it with a so-called “mock object” (“Service Stub pattern” [PEAA 02]) that simulates the real object. In other words, when testing your client you can replace the real “ArchitectureA_Adapter” with another class that has the same interface (for example a subclass) to enable testing of most of the code even if you currently do not have a network connection.

1.1.9 Related patterns

This “Architecture Adapter pattern” (AAP) is used by the service-oriented architecture pattern to help with the implementation transparency.

The “SOAP engine” (e.g. “Apache Axis”) that is mentioned above is described in the “Web Services Gateway pattern” [PLoP 02].

AAP is similar to the ”Channel Adapter pattern” [EIP 03].

In the “Forwarder-Receiver pattern” [POSA 96], the “Forwarder” class roughly corresponds to the client-sided “ArchitectureA_Adapter” above, while the “Receiver” corresponds to the server-sided SOAP engine. However, that point of view is a little bit too simplified and a more correct (but also more complicated) comparison would be to say that both endpoints of the AAP, i.e. the “ArchitectureA_Adapter” and the SOAP engine might instead contain a forwarder and a receiver. The AAP is classified as an architectural pattern while the “Forwarder-Receiver” is a design pattern described in a more detailed level saying that the unmarshaling is a responsibility of a receiver class while a forwarder class will do the marshaling. In other words, the generated code for the “ArchitectureA_Adapter” might use a generated “forwarder” as a helper object that takes care of marshaling the request to a SOAP message, and it might later use a generated “receiver” object to unmarshal the SOAP response, instead of generating all code into the “ArchitectureA_Adapter” class itself.

The “Service Gateway” pattern [ESP 03] is similar to the client part of the AAP, i.e. the “ArchitectureA_Adapter” object above, and defines a gateway object that encapsulates the network communication code. The “Service Interface” pattern [ESP 03] is similar to the server part of the AAP.

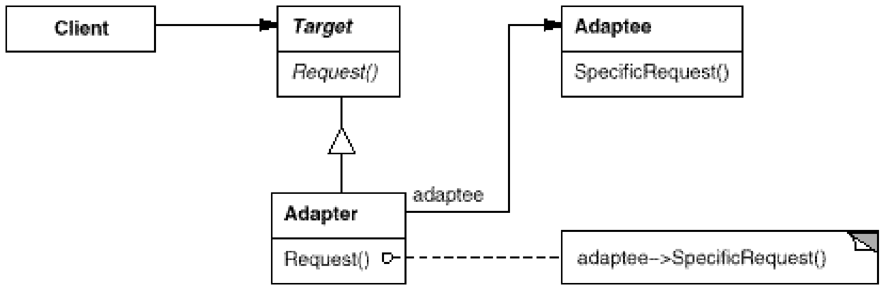
The “Web Service Broker” [CJP 03] is another name of a pattern that describes the server part of the AAP, i.e. how to take care of the XML in a SOAP invocation and then delegate to business service objects that may be reused from other non-web service contexts.

The server part of the AAP is also described 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.

The AAP is similar to the GoF Adapter pattern, which is illustrated in the diagram below. Compare the Adapter diagram below to the conceptual diagram above and you may notice the following corresponding classes/interfaces:

- Client = ArchitectureA_Adapter
- Target = “CommonArchitectureInterface”

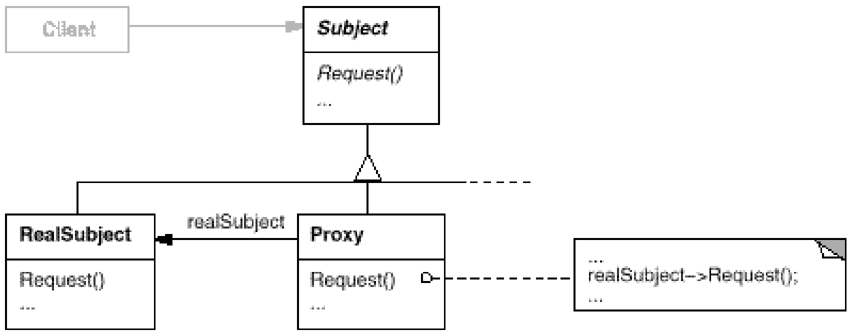
- Adapter = ArchitectureB_Adapter
- Adaptee = ArchitectureB_Class



However, as discussed above, "CommonArchitectureInterface" is not an actual interface, but rather a SOAP engine communicating with clients with an SOAP interface defined in XML with WSDL.

The AAP also somewhat resembles the GoF proxy pattern. More specifically, the "remote proxy" is described by GoF as a class that "provides a local representative for an object in a different address space". If you only consider this definition then the architecture adapter seems to be using the remote proxy pattern. However, if you study the proxy pattern in more detail and compare the GoF proxy class diagram below to the AAP diagram above you can see that the structure of the classes/interfaces is not very precisely corresponding:

- Client = ArchitectureA_Class
- Subject = "Not applicable"
- Proxy = ArchitectureA_Adapter
- RealSubject = ArchitectureB_Adapter



As you can see, the significant interface "Subject" in the GoF proxy pattern is not being used, because there is no such common interface in the architecture adapter pattern, which of course is not possible since adapter A and B (Proxy and RealSubject) may be programmed in different languages and therefore can not have a common programming language native interface but can only use the same XML based interface defined with WSDL.