1.1 Service Directory Pattern

1.1.1 Name and Source

Service Directory Pattern
Page 75-97 in the book "Web Service Patterns: Java Edition" [WSP 03]

1.1.2 Also Known As

-

1.1.3 Type

Architectural design pattern.

1.1.4 Intent

To provide location transparency.

1.1.5 Problem

Web Services is a Service Oriented Architecture (SOA) and one of the intents of a SOA is to provide location transparency, which means that you want to be able to dynamically locate and immediately start using a new component that implements the interface that your application uses, without first being forced to modify your code with the URL for the newly found Web Service.

How can location transparency be achieved with Web Services ?

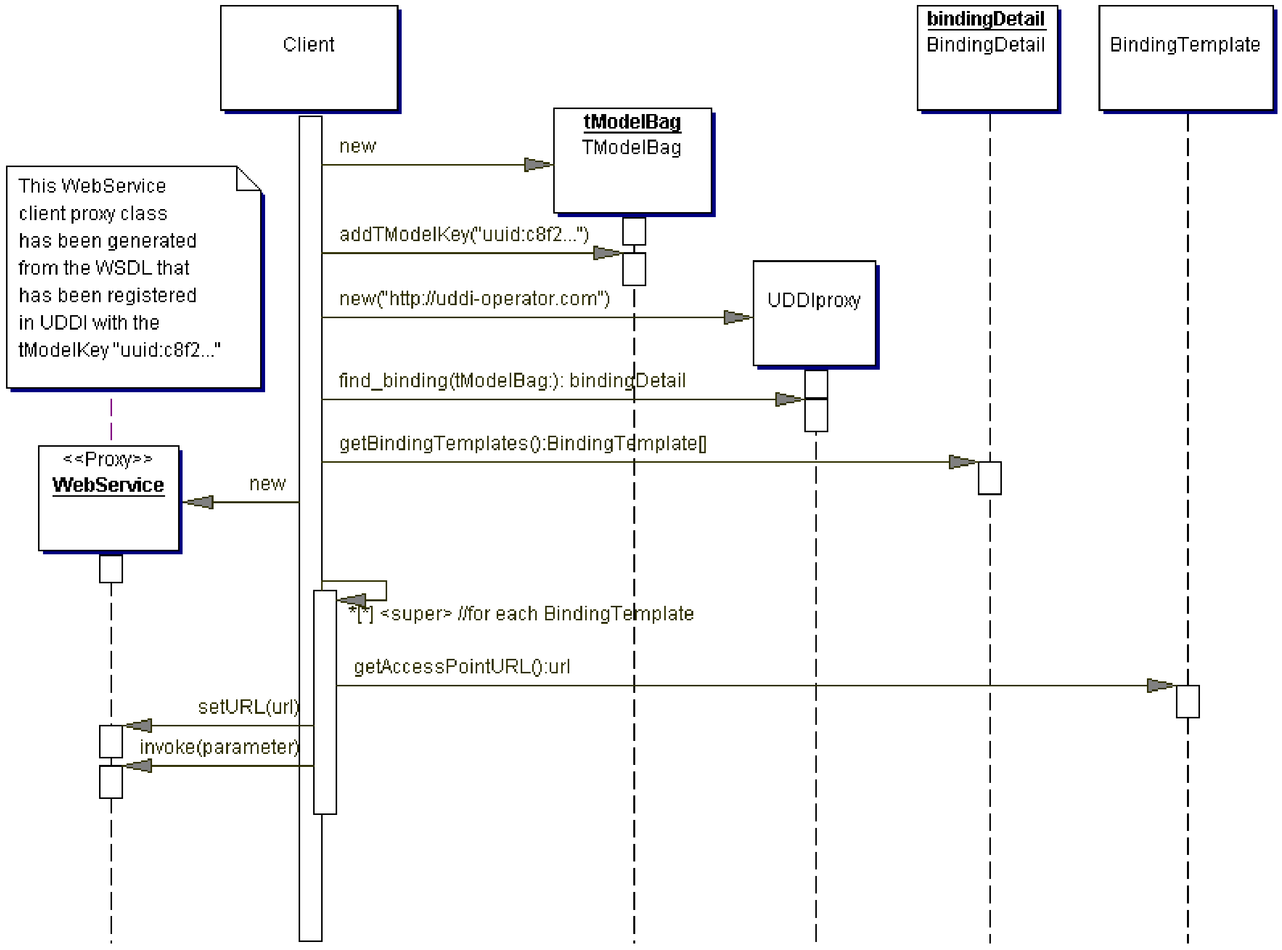
1.1.6 Forces

1.1.7 Solution

The solution assumes that the interface your application is using have been registered as a WSDL interface in an UDDI tModel structure, which refers to the WSDL file in the <overviewURL> element within the <tModel>.

When you have generated a proxy class that implements a certain WSDL interface described with a tModel in UDDI, then you can query an UDDI operator for all Web Services that implement that interface, and you can iterate through them, and for each iteration you get the URL for the Web Service and then sets that URL to the proxy class (if the generated proxy class provides some kind of “setURL” method) and invokes the method you want to invoke to the web service. Consider the “Web Service Interface” pattern if the generated proxy can not be reused for different web service implementations.

The important thing to note in the diagram below is that neither the “Client” class nor the “WebService” proxy class are using any static URL for invoking the WebService, but the actual URL’s for the Web Services implementations (that are implementing the WSDL interface used by the application) being used are instead found dynamically in runtime by querying an UDDI server through a proxy object.


Client
tModelBag
TModelBag
bindingDetail
BindingDetail
BindingTemplate
This WebService client proxy class has been generated from the WSDL that has been registered in UDDI with the tModelKey "uuid:c8f2..."
new
addTModelKey("uuid:c8f2...")
new("http://uddi-operator.com")
UDDIproxy
find_binding(tModelBag:): bindingDetail
getBindingTemplates():BindingTemplate[]
<<Proxy>>
WebService
new
*[*] <super> //for each BindingTemplate
getAccessPointURL():url
setURL(url)
invoke(parameter)

[the occurrence of "<super>" in the diagram is not intentional and will be removed in the final version]

- WebService – A proxy object for a Web Service that is generated from a WSDL interface as described in the “Architecture Adapter pattern”. Depending on the tool that generates this proxy class, maybe there will not be any “setURL” method that can change the URL, but you rather might have to instantiate a new object for each bindingTemplate in the iteration and supply the url as a parameter to the constructor of the proxy class, or maybe the url is only hardcoded when the class is generated. In that case one possibility is to use an interface for the web service. Refer to the section “related patterns” regarding this issue.
- Client – An object that wants to invoke the Web Service method “invoke(SomeDataType parameter)” which is a method that is described in a certain WSDL file that has been registered as a UDDI tModel that refers to the WSDL. In this example, the Client object invokes that method once for every Web Service in the UDDI registry that has been defined as implementing this WSDL/tModel, but another option would be to just invoke the method for the first Web Service found, instead of iterating through all of them.
- tModelBag – An object representing the UDDI structure <tModelBag> which is a structure that can be used as container for tModel parameters in UDDI queries. The example diagram illustrates that you wants to search for Web Services that implements

the WSDL interface that has been registered as a tModel with a certain tModelKey, which is some unique UUID.

- UDDIproxy – An UDDI server is itself a Web Service with a SOAP API, and this UDDIproxy class is a proxy class for the SOAP methods defined by UDDI, and it could be generated from the WSDL describing the UDDI web service, just like any WSDL file can generate proxy classes, as described in the Architecture Adapter pattern. There are also some existing class libraries with class structures similar to the UDDI xml structures, for example UDDI4J is a UDDI class library for java that has a class "UDDIProxy" (and UDDI4J also has the other three classes to the right of the Client class in the diagram).
- BindingDetail – An object representing the UDDI structure <bindingDetail> which is the structure that gets returned when invoking the SOAP method <find_binding> which is done in the diagram through the UDDI proxy method with the same name.
- BindingTemplate – An object representing the UDDI structure <bindingTemplate> that is one of the four core UDDI structures, that contains an "accessPoint" which is a URL you can use for invoking the Web Service.

1.1.8 Consequences

It is probably not very likely in a real world scenario that you would want to use the pattern for automatically start doing business with unknown companies just because they have added themselves in the global UDDI Business Registry and submitted a web service that implements the interface your are searching for in UDDI. It is more likely that you will extend the pattern so that your application only uses those Web Services that has been manually approved. You may also want to subscribe for tModels and let an UDDI registry send you notifications when new Web Services with a certain interface has been added to the registry. Another option, which would not require any extra application logic, is to use your private UDDI registry that only contains approved companies, when your application is doing lookups.

1.1.9 Related patterns

This "Service Directory pattern" is used by the "Service-Oriented Architecture pattern" to help with location transparency.

It is generally better to implement the "Service Factory Pattern" instead of this pattern, since that pattern avoids the low-level UDDI details used from within the Client objects in this pattern.

The "Client-Dispatcher-Server pattern" [POSA 96] (CDS) is a generic communication pattern that just like this "Service Directory pattern" provides location transparency, and just like the above mentioned "Service Factory" it seems as if the CDS also wants to isolate the lookup code by saying '*The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers*' [POSA 96, page 324]. However, the "Service Factory" is indeed a pattern that provides an extra layer within the client but when I look at the CDS sequence diagram (quite high-level) the "Dispatcher" class corresponds to the UDDI web service that often resides in another process boundary and thus does not correspond to the client-sided class "ServiceFactory" as you might expect from the above quoted statement. Thus I would rather say that the CDS is more similar to this "Service Directory pattern" since it does not provide the extra client layer that the "Service Factory" does to separate the code that establishes the connection. A difference between "Service Directory" and CDS is that the remote "Dispatcher" object in the CDS tries to establish a communication link with the server, while the UDDI object in this "Service Directory" does not try to connect to the web service.

In the sequence diagram above, the generated proxy class is assumed to provide a “setURL” method that enables the proxy object to be reused for different implementations of the same web service. However, some WSDL tools maybe do not generate such methods but only generates a class from a WSDL that includes a hardcoded URL as defined in the WSDL file. Another option than to reuse the same proxy class with a “setURL” method is to use an interface from the client that is implemented by different implementations with hardcoded generated URL’s. Refer to the “Web Service Interface” for a pattern about how to use refactoring to create such an interface if it is not generated by the WSDL tool.