1.1 Observer Pattern for Web Services

1.1.1 Name and Source

Observer pattern ("for Web Services" has above been added to the name by the author of this thesis document, to distinguish it from the GoF observer pattern).
Page 187-204 in the book "Web Service Patterns: Java Edition" [WSP 03]

(The basic Observer pattern was published in [GoF 95])

1.1.2 Also Known As

-

1.1.3 Type

1.1.4 Intent

To let the server provide a notification mechanism that, for example, can inform a client when a long-running and asynchronously invoked web service has finished.

1.1.5 Problem

When a web service client has started an asynchronous business process it will later want to know when the business process has completed and of course also the result of the process. If the EventMonitor pattern is used it might cause a lot of unnecessary network traffic if the polling frequency is high, and in some situations it really is necessary to get immediate notification about some change. If possible, it would generally be a better solution to receive the notifications immediately. Except from sending notifications about a long-running business process, there are also other situations where the pattern can be used, for example UDDI 3 uses the observer pattern to notify about changes to the directory, e.g. changes in tModels.

How can a web service client immediately become informed about some event, for example the result of a long-running business process ?

1.1.6 Forces

1.1.7 Solution

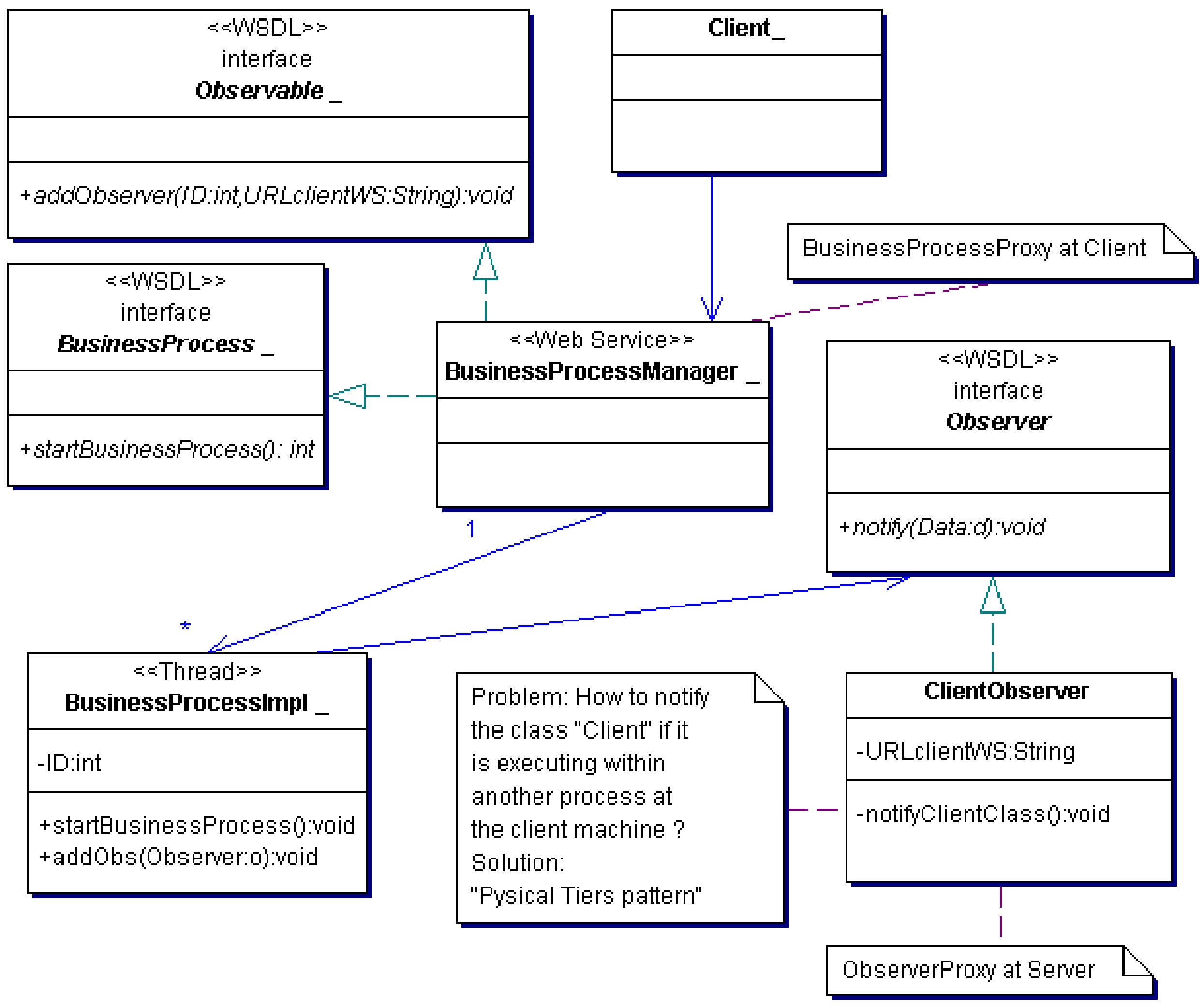
The solution assumes that the client itself can provide a web service, e.g. by using the "Faux Implementation pattern".

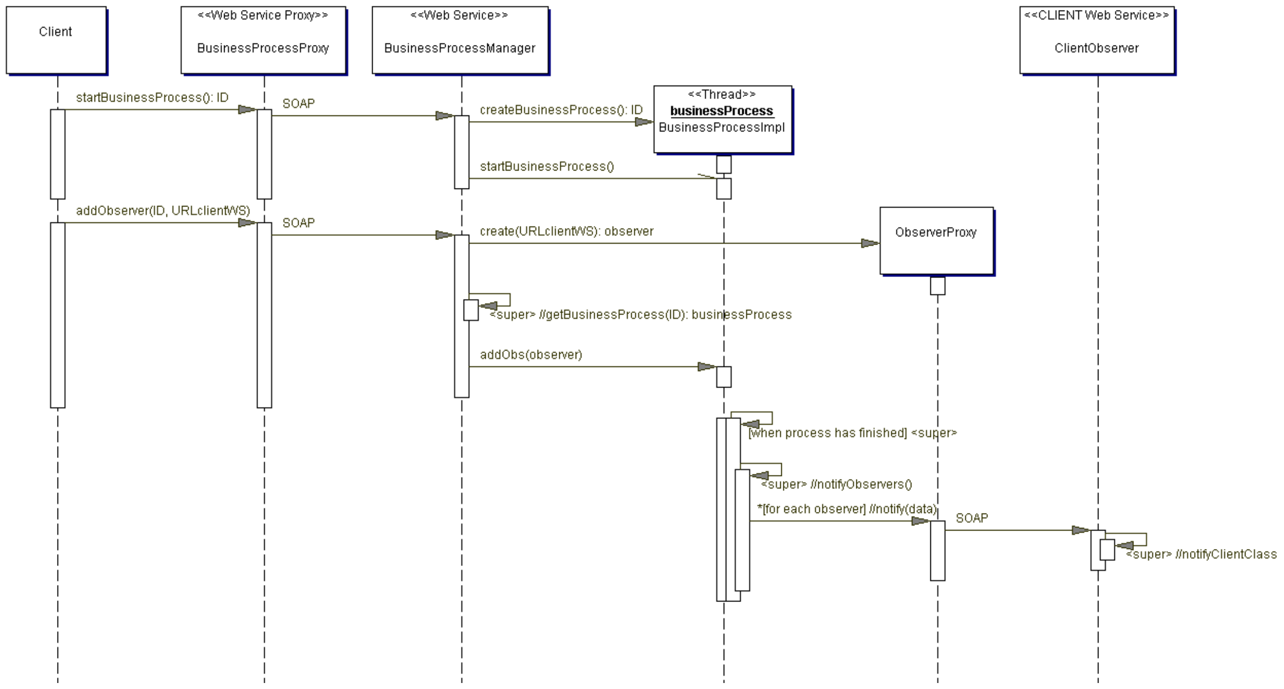
You can define an observer interface with WSDL and use it to generate an Observer proxy class to be used by the web service that will be observed. The web service client will then itself have to provide a web service that implements this WSDL interface to get notifications from the server. The business process manager web service (described in the "Asynchronous Business Process pattern") will also implement an Observable ("Subject" as in [GoF 95]) interface but it should be noted that instead of receiving a remote reference to an Observer, the "addObserver" method will receive an URL to the client web service implementing the Observer WSDL. The reason is that SOAP messages are stateless and therefore remote references can not be used. Instead of an URL an email-address might be used to send SOAP notifications with SMTP instead of HTTP. Except from the URL, the "addObserver" method also receives a parameter with the ID that can be used to identify the process or event that the client is interested in.

The class diagram below should be considered as a conceptual diagram rather than a real implementation diagram, since it includes classes at both the client and the server. In fact, the diagram includes server classes plus classes executing in two (possibly) different processes within the client machine. The “Client” class may represent a stand-alone application while the “ClientObserver” may be executing in a different client process in a web service engine that somehow will need to communicate back to the “Client” class (see the “Physical Tiers pattern”) that was the class in the stand-alone application that wanted to get the notification. A stand-alone application might also implement a web service as in the “Faux implementation pattern”.

For a real implementation you probably would like to create different class diagram for the client and server, but for the purpose of illustrating this pattern this class diagram should suffice together with the details in the sequence diagram below, and the following comments about the diagrams:

The interfaces below with the stereotype “WSDL” are interfaces corresponding to the interfaces generated from “portType” element in a WSDL file, or interfaces extracted from a web service proxy object as described in the “Web Service Interface” pattern. These language specific (e.g. Java or C#) interfaces are generated from the WSDL file, at least for the client, while the interfaces at the server side may have been used for generating the WSDL. The interfaces are implemented by a language specific class (generated at the client) at both sides and both of them are illustrated in the sequence diagram but in the class diagram, the “BusinessProcessManager” represents both the server sided “BusinessProcessManager” and the client sided proxy “BusinessProcessProxy” in the sequence diagram. The client sided “ClientObserver” in the class diagram does also represent the server sided “ObserverProxy” in the sequence diagram.


<<WSDL>>
interface
Observable _
+addObserver(ID:int,URLclientWS:String):void
Client _
BusinessProcessProxy at Client
<<WSDL>>
interface
BusinessProcess _
+startBusinessProcess(): int
<<Web Service>>
BusinessProcessManager _
<<WSDL>>
interface
Observer
+notify(Data:d):void
1
*
<<Thread>>
BusinessProcessImpl _
-ID:int
+startBusinessProcess():void
+addObs(Observer:o):void
Problem: How to notify the class "Client" if it is executing within another process at the client machine ? Solution: "Pysical Tiers pattern"
ClientObserver
-URLclientWS:String
-notifyClientClass():void
ObserverProxy at Server


Client
<<Web Service Proxy>>
BusinessProcessProxy
<<Web Service>>
BusinessProcessManager
<<CLIENT Web Service>>
ClientObserver
startBusinessProcess(): ID
SOAP
createBusinessProcess(): ID
<<Thread>>
businessProcess
BusinessProcessImpl
startBusinessProcess()
addObserver(ID, URLclientWS)
SOAP
create(URLclientWS): observer
ObserverProxy
<super> //getBusinessProcess(ID): businessProcess
addObs(observer)
[when process has finished] <super>
<super> //notifyObservers()
*[for each observer] //notify(data)
SOAP
<super> //notifyClientClass

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

BusinessProcessProxy – An architecture adapter representing the web service.

BusinessProcessManager –An implementation of a web service interface. It implements a business process, but also an observable interface with an “addObserver” method. This manager starts a business process in a separate thread with an asynchronous invocation, as described in the “Asynchronous Business Process pattern”.

BusinessProcessImpl – The business process object executing in a long-running thread started by the process manager. Note that this business process class does not need to have a method named “addObserver” as in the WSDL interface, because it is an internal implementation detail and therefore could be named to anything, for example “addObs” as illustrated above.

ObserverProxy – This proxy object is generated from the observer WSDL. It is executing at the server and is a proxy for the “ClientObserver” web service implemented by the client.

ClientObserver – This is an implementation of the web service observer described with WSDL. This web service is provided by the client that wanted to get notifications from the “BusinessProcessManager” (Observable) web service. This web service might execute within the same computer process as the “Client” and “BusinessProcessProxy” objects, for example it does so if the “Faux Implementation pattern” is used. However, it may also execute outside of those objects process and in that case the “ClientObserver” somehow needs to communicate the notification back to the “Client” and that may be done with the “Physical Tiers Pattern”.

1.1.8 Consequences

Interprocess communication may be necessary to implement within the client computer, as described in the related patterns section.

The notifications can be sent with HTTP but also as email with SMTP.

It is difficult to predict all kind of data to supply as parameter in the “Observer.notify” method. Therefore it may be necessary for the observing client to call the web service again to get more detailed data when some update has occurred. Another option might be to use a very generic parameter for the notify method, e.g. XML content. However, the XML option may require more low-level programming/parsing compared to simply using a strongly typed object that is automatically generated from the WSDL. You can also use the “Partial Population Pattern” for a generic data object that is more easy to use compared to XML parsing.

If you expose a business object or collection as a web service, you should be aware that the observer pattern only works if the target objects already exist. In other words, if you want to observe creation as well as deletions and updates of business objects, then you should observe the business object as well as the collection of business objects.

1.1.9 Related patterns

The “Publish/Subscribe pattern” is similar to the “Observer pattern” which is described in the problem section in the “Publish/Subscribe pattern”.

The pattern typically needs to be combined with the “Physical Tiers Pattern” or the “Faux Implementation pattern” since the client object that originated the request may not necessarily be a web service itself that can listen for incoming SOAP notification messages, nor be a POP server listening for incoming SMTP messages.

The "Asynchronous Business Process pattern" describes the invocation of a long-running Web Service without locking the client process, while this "Observer pattern" describes how the client later can be notified when the long-running Web Service has finished.