1.1 Event Monitor Pattern

1.1.1 Name and Source

Event Monitor Pattern
Page 169-186 in the book "Web Service Patterns: Java Edition" [WSP 03]

1.1.2 Also Known As

-

1.1.3 Type

Micro-architectural design pattern

1.1.4 Intent

To get notified about when a long-running, asynchronously invoked web service has finished, if the web service does not provide a notification mechanism or if the client itself can not receive SOAP notifications.

1.1.5 Problem

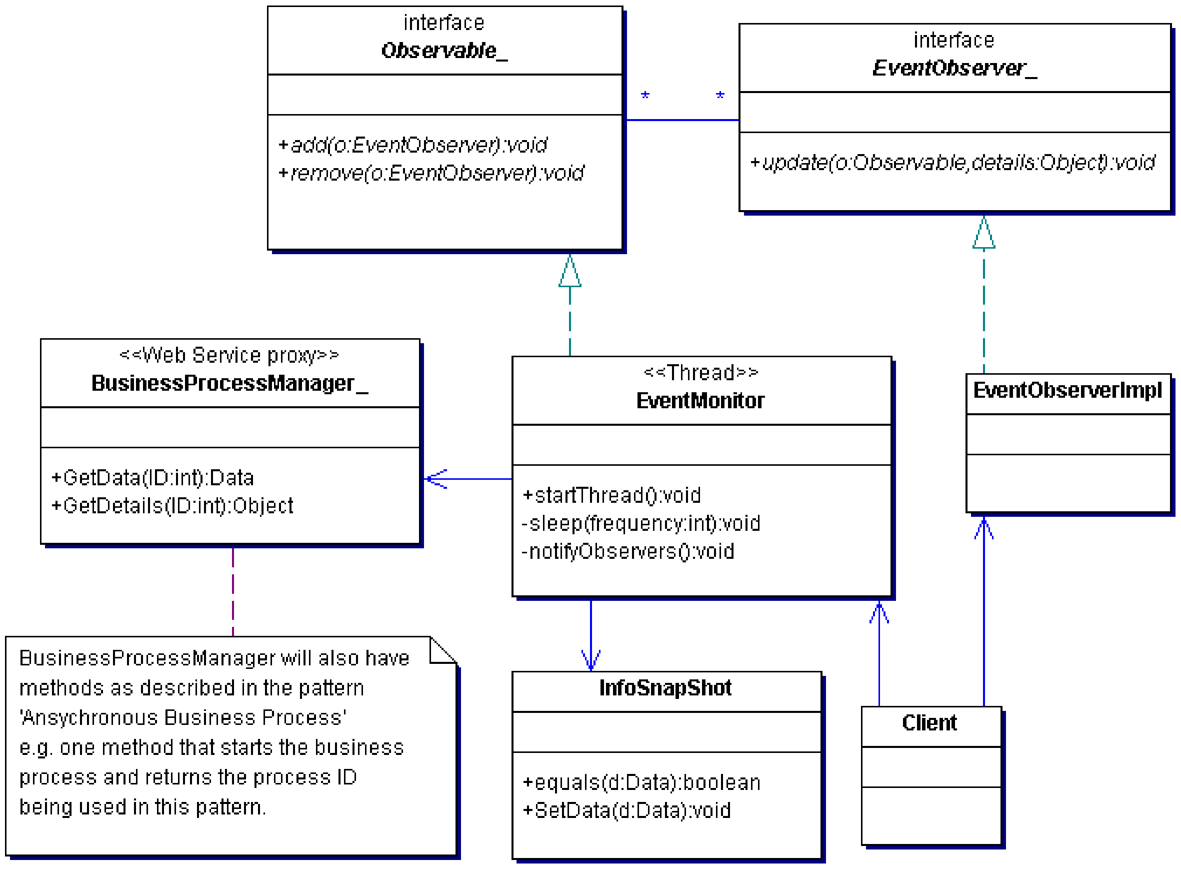
When a web service client has started an asynchronous business process it will later want to know when the business process has completed and of course also the result of the process. The optimal way of getting this information is that the web service will provide a notification mechanism that sends a message to the client with all information that the client is interested in. However, this solution is not always feasible, since the web service may not provide a notification mechanism and it may be difficult to persuade the service provider to modify their existing applications to implement web service notification mechanism. Even if the server does provide a notification mechanism, it may not support the right kind of detailed information that the client wants, since it may be difficult to predict all kinds of information that clients might be interested in. Another problem is that a notification mechanism will require the client to also provide a web service that can receive messages about the completion of the business process.

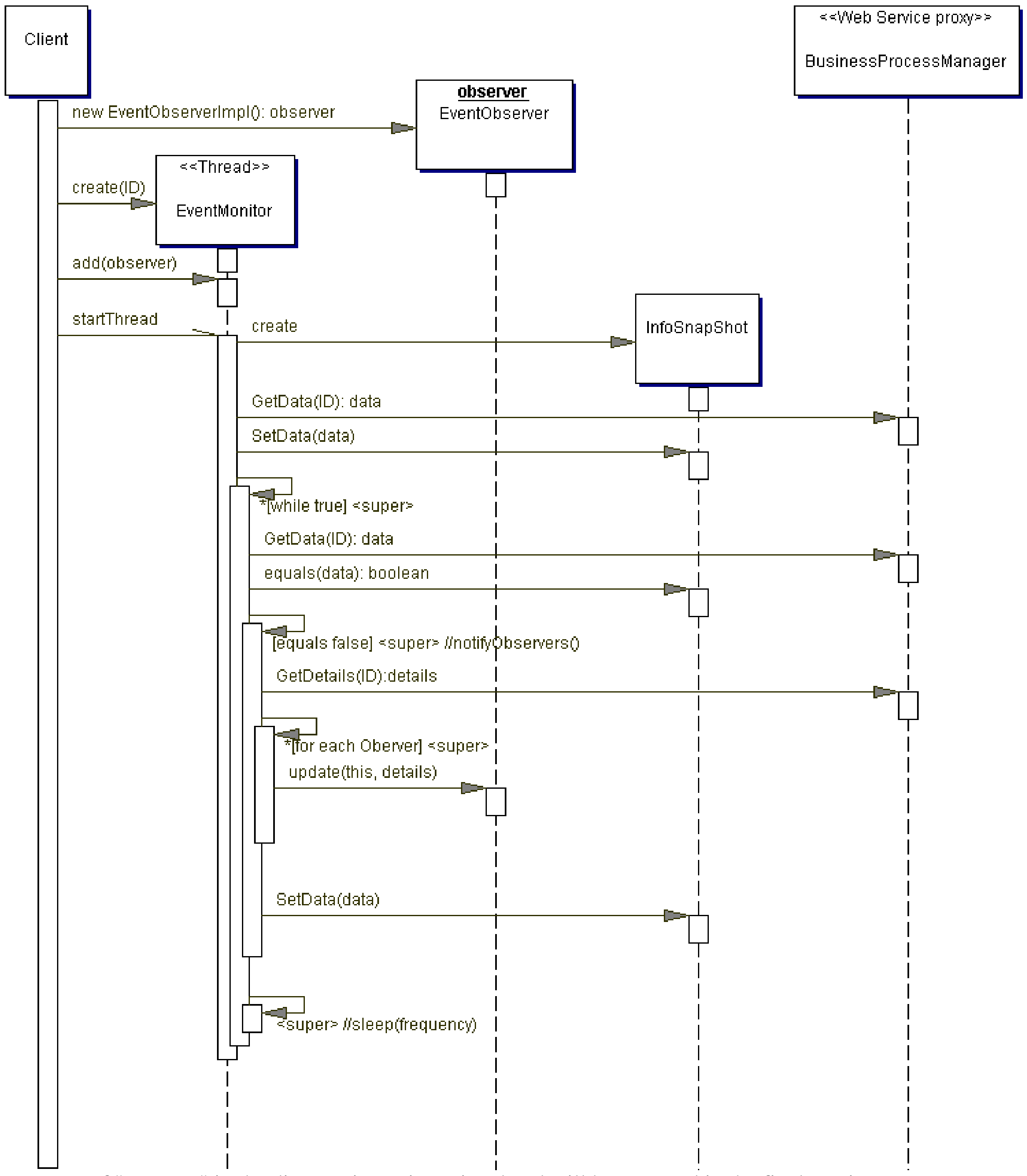
How can a web service client become informed about the result of an asynchronous business process if the web service does not provide a notification mechanism or if the client cannot act as a web service itself to receive the notifications sent as SOAP messages ?

1.1.6 Forces

1.1.7 Solution

The client application creates an "Event Monitor" that will run in a separate thread and it continuously invokes the web service to check for changes that are expected to occur as a result of the asynchronous method call. When the "Event Monitor" detects a change it notifies the “event observers” that been registered with the "Event Monitor".


interface
Observable_
+add(o:EventObserver):void
+remove(o:EventObserver):void
*
*
interface
EventObserver_
+update(o:Observable,details:Object):void
<<Web Service proxy>>
BusinessProcessManager_
+GetData(ID:int):Data
+GetDetails(ID:int):Object
<<Thread>>
EventMonitor
+startThread():void
-sleep(frequency:int):void
-notifyObservers():void
EventObserverImpl
BusinessProcessManager will also have methods as described in the pattern 'Ansychronous Business Process' e.g. one method that starts the business process and returns the process ID being used in this pattern.
InfoSnapShot
+equals(d:Data):boolean
+SetData(d:Data):void
Client



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

EventMonitor – The object that is running in a thread and continuously invokes the web service.

EventObserver – This interface is implemented by the objects that want to get notified about when the observed data has changed.

InfoSnapShot – A helper object for the EventMonitor that encapsulates the data returned from the previous web service request and also the logic for comparing the previous data with the latest data received from the web service.

BusinessProcessManager – A client proxy to the web service that, in the sequence diagram above, is assumed to already have been started and returned a process ID, as illustrated in the sequence diagram for the "Asynchronous Business Process pattern".

1.1.8 Consequences

The appropriate polling frequency may be hard to too predict, and it should be possible to configure the frequency, i.e. do not hardcode it. If the frequency is too high there will be a lot of unnecessary network traffic. On the other hand, if the frequency is low it may take a lot of time to discover the new data, but if it is very important to always have the latest data then the EventMonitor should not be used. In some applications though, it may be enough to poll once per 24 hours and in such cases the EventMonitor would be an option to consider.

The "EventMonitor" should usually not be the first choice when implementing new applications with a web service interface, but rather it should be considered as a last resort when integrating existing applications that originally were not designed and implemented to provide notification mechanisms or when the client itself cannot act as a web service to receive the notifications.

If multiple client objects will want to observe the same change from the web service, then the network traffic can be reduced by letting these client objects, i.e. EventObservers, reuse the same EventMonitor to receive notifications instead of letting many EventMonitor objects keep invoking the web service.

The EventMonitor pattern can be useful when it is difficult to add a notification mechanism in an EAI (Enterprise Application Integration) scenario, for example when customer data need to be replicated between an existing CRM (Customer Relationship Management) system and some other existing ERP (Enterprise Resource Planning) system. Then a web service interface can be added to both of these systems that can be used by an EventMonitor to detect that data has changed in one system and that the other therefore also has to be updated. Adding such interfaces can be easier than implementing notification mechanisms in the existing systems.

1.1.9 Related patterns

The EventMonitor uses the traditional Observer pattern [GoF 95]. However, since this thesis document also describes a pattern that is named as Observer (described in the same book as EventMonitor, but I added the suffix "for Web Services" to that Observer pattern name) it should be noted that the EventMonitor is a pattern where both the GoF Subject and Observer (Observable and EventObserver) are client objects, while the "BusinessProcess" in the "Observer pattern for Web Services" is the GoF Subject and is a web service.

The InfoSnapShot class can be subclassed (or be an interface), if you would want to switch between different implementations of how to determine changes, and that would be an application of the Strategy pattern [GoF 95].

The "Asynchronous Business Process pattern" describes the invocation of a long-running Web Service without locking the client process, while this "Event monitor pattern" describes how the client then can figure out when such a business process has finished if the client is not able to receive a notification from the long-running Web Service when it has finished.