1.1 Business Object Pattern

1.1.1 Name and Source

Business Object Pattern
Page 99-114 in the book "Web Service Patterns: Java Edition" [WSP 03]

1.1.2 Also Known As

"Business Object" is also used as a pattern name in the book [CJP 03] which is discussed in the section below with related patterns.

1.1.3 Type

Micro-architectural design pattern.

1.1.4 Intent

To enable changing of a property for a business object exposed as a web service, when the property is a non-primitive property, considering that SOAP messages are stateless.

1.1.5 Problem

A business object is representing some concept from the real world, e.g. company, customer or product. The attributes that are aggregated within a business object may be primitive, i.e. strings or numbers, but attributes may also be instances of other classes. These classes may also have their own attributes that may be primitive as well as instances of classes. In other words, a business object may contain many levels of nested non-primitive objects.

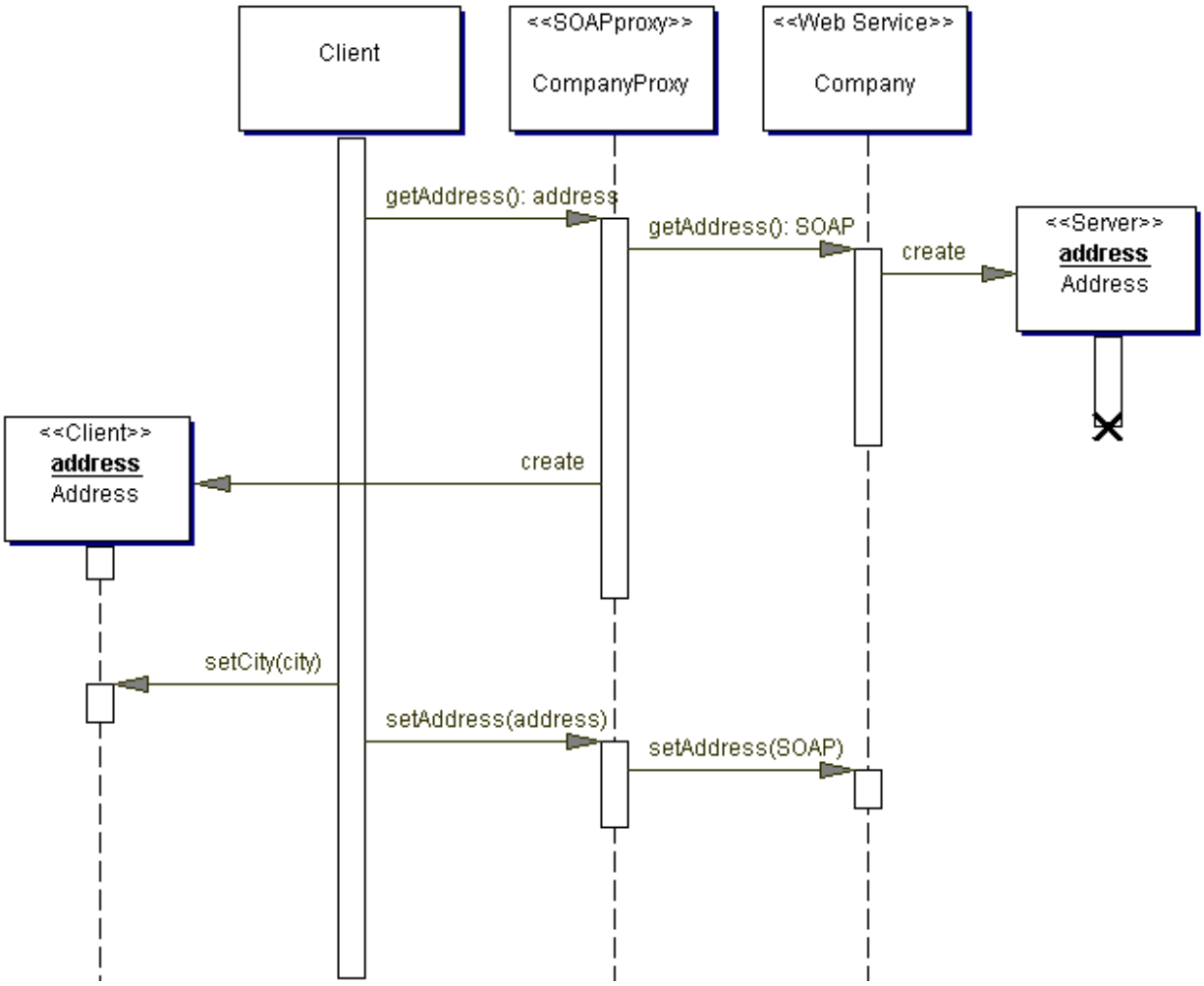
How can you change a non-primitive property associated with a business object (BO) if you want to expose the BO as a Web Service, considering the stateless nature of SOAP messages ?

1.1.6 Forces

1.1.7 Solution

Note that the problem does not apply when you want to change a primitive property that directly belongs to a Web Service exposed business object, because then you would be able to change it by simply calling the corresponding method of the proxy object. For example, consider a business object representing a company being exposed as a business object, and suppose you want to change the name property, which is just a primitive string. Then you just would make a method call like setName("The Name of My Company") on the company proxy object. Further, if you would choose to let the entire address of the company also be represented as a primitive string attribute, then you could do a similar method call to change the address. However, if you are using object-oriented business objects it is more likely that you would have an object model where the company object aggregates an address object that contains primitive strings like city, street, state and so on. When you are dealing with such an object model locally in a typical object-oriented language that uses references, then you can just get a reference to the address object and then invoke methods on that reference for doing changes of the primitive properties for the address, and after that you do not have to do anything more. However, when dealing with web services, the objects that can be replicated on the client are only copies of the data sent over SOAP. For example, the address object in the diagram below is not a proxy object forwarding any method invocations to the server with SOAP. The address object is just an object that is

entirely local within the client and it is created in the getAddress method in the company proxy class after it has parsed the SOAP response message. Therefore, the important thing to note in the diagram below is the setAddress method call, which is needed to update the server with the new city in the address.



Address<<Client>> – This address class is located at the client and contains copy of the data in a corresponding address object at the server, and that data is sent to the client in a SOAP message.

Client – A client class that first calls the business object proxy (CompanyProxy) to get address data from a SOAP message, and then changes the city property in the local object, and then submits the change of data back to the server through the proxy object.

CompanyProxy – A client business object that is a proxy object communicating with a server sided business object exposed a web service. It converts local client objects to and from the XML data in SOAP messages.

Company – A business object exposed a web service.

Address<<Server>> – This address class is located at the server and is included in the diagram to illustrate the stateless nature of SOAP. An instance of this class only lives during a SOAP method invocation and the object exists in the server only, while the class Address<<Client>>

will not be a remote reference to this class, but it will only replicate the data within the server sided class.

1.1.8 Consequences

Generally, you should not want to use this pattern. Therefore, from one point of view, it could be considered as an anti-pattern rather than a real pattern. However, **if** you really want to expose a BO as a web service, then the pattern does provide a simple solution for how to change non-primitive BO properties.

The pattern has these consequences:

- The pattern couples a BO with a Web Service, and therefore you may get problem if you want to delete the BO. It would then be better to use the Business Object Collection pattern, which lets you remove a BO from the collection.
- If you are exposing an object-model to clients then some clients using non-object-oriented languages may not be able to use your web service.
- Exporting the object model to the clients will make it more difficult for you to change the object model in your implementation without affecting clients because they will have created dependencies to your object model from their applications.

The fact that all client objects will not be remote references but rather local copies is not a really big problem but might be considered as a good thing since it more or less forces you to try to do a better design than you might be tempted to do if it was easy to expose entire object models to the client and let changes be automatically replicated to the server. For example, assume that a company BO would aggregate a “Boss” object, which aggregates an Address object, and that you would want to change the city property in that Address object. Then if you would be able to use remote references you might be tempted to expose many objects to the client and let them use code like this:

company.getBoss().getAddress().setCity(“my city name”);

However, if that code would work (which it does not) when “company” is a web service proxy, it would make the client depending on many remote classes, i.e. it would lead to “High Coupling” and also violate the “Law of Demeter” (aka the more intuitively descriptive name “Don't Talk to Strangers”) which are GRASP (General Responsibility Assignment Software Patterns) that indicate bad design if they are not used. Therefore, it does not matter very much that the code above would not work at the SOAP client, because you should not be using it anyway. Instead, try to use the “Business process pattern” and try using a more flat interface for web services rather than exposing an object-oriented model to clients.

1.1.9 Related patterns

Another solution to the described problem is to extend the business object and let it also contain such primitive types that belongs to, and will be delegated to, other aggregated classes. This solution is similar to the solution of the “Composite Entity pattern” (Previously known as “Aggregate Entity”) which is a J2EE (Java 2 Enterprise Edition) pattern [CJP 03] where an EJB (Enterprise JavaBean) will represent and expose the properties of many underlying entity EJB’s.

This “Business Object pattern” is similar to the “Business Object Collection pattern” since a business object collection may be considered as a business object that contains other business objects as the non-primitive objects being discussed in this “Business Object pattern”.

Usually it is better to use the “Business Process pattern” instead of exposing a business object as a web service. “Build Component-based Software Architecture” [PLoP 02] and “Large Grained

Service" [PLoP 02] are some of the patterns that tell you to expose large-grained business process objects instead of business objects as web services.

The book [CJP 03] also defines a pattern named "Business Object" (BO). In my opinion, the BO is more a concept than a general solution to a recurring problem (the definition of a pattern). The bottom line of the BO [CJP 03] simply seems to be that it is better to use objects with business logic than to use a procedural programming approach when you have a conceptual domain model. Just like (virtually) everyone else, the BO [CJP 03] pattern does not recommend to expose a business object to remote clients. Regarding the above described BO [WSP 03] I actually think it is very unclear exactly what the general problem is, and since I do not think it would be a pattern to simply define what a business object means I instead described the most interesting part of the business object chapter, i.e. that you should be aware of the fact that **if** a business object is exposed as a web service then it will be stateless.