

The SOA Magazine

Feature Article



Service-Orientation vs. Object-Orientation: Understanding the Impedance Mismatch

by Larry Guger

Published: July 18, 2008 (SOA Magazine Issue XX: July 2008)

[Download this article as a PDF document.](#)

Abstract: Object-oriented programming languages and techniques provide a powerful means for designing and building applications. These techniques do not always translate well into a service oriented paradigm. Service orientation demands a different set of design guidelines and requirements than an object-oriented application. Understanding how an object-oriented design can negatively impact a service-oriented design is key to building services that support an agile enterprise. This article examines where the two designs impact each other as well as methods for addressing the incompatibilities between the two while still leveraging the power of both.

Introduction

Object orientation is a good thing. I would like to believe that I write code in a well-defined object oriented manner, taking advantage of all the goodness that is provided, such as encapsulation, polymorphism and inheritance. These are important concepts that make modern software applications easier to develop, enhance and maintain. I'm sold.

Service-orientation is a good thing too. As the industry moves toward service-orientation we naturally take along a lot of what we have learned over the years and apply this to the new way of doing things. Visual Studio, the .NET framework, and especially Windows Communication Foundation (WCF) support the development of service-oriented applications. This was one of the core design goals behind WCF, but moving from object-oriented design techniques to service-oriented design techniques is not without its challenges.

Part of the reason can be apportioned towards the tooling. We have very mature tools to support object-oriented design and development but fewer tools that emphasize service-oriented design. This is partly because we, as an industry, are still really figuring out what service-orientation really means. This article explores what I am referring to as the "impedance mismatch" between the two design paradigms.

Note: Most of the concepts and ideas presented are not specific to .NET or Visual Studio until I refer to the namespace generation problem later in the discussion (as it manifests itself in Visual Studio). However, it is worth noting that this problem can present itself on any platform.

Designing an OO System

Here is an example of one model that would make sense in the object-oriented world:

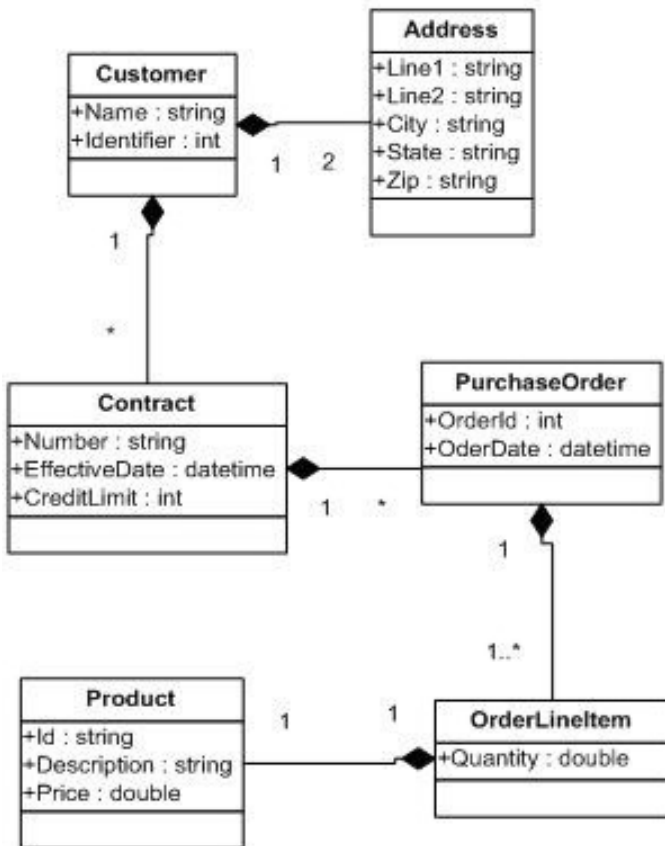


Figure 1

Figure 1 is a simplified model designed to support some form of purchasing functionality provided by the application. A customer contains a mailing address and a shipping address and the customer can also hold many contracts with the company that is supplying products.

The customer is able to place orders against these contracts with any given order containing one or more line items each of which is an order for a product. In addition, the model permits the developer to navigate from a PurchaseOrder object to the Contract under which the order was placed by using an object reference. Likewise, a Contract will contain a collection of all of the PurchaseOrders placed under it. This model is also repeated between the Customer and its Contracts as well as PurchaseOrders and OrderLineItems. We now have a nice object model that permits navigation between related objects in any direction.

Service Enablement

In a distributed computing environment such as is found in almost every enterprise today there is a business logic/service tier that resides on some central server farm that exposes services for working with the contained business functionality. This service tier is accessed by a client tier, whether the client is a Web application, a rich client application or a B2B service implementation. To support the object model above, one can imagine a collection of services that are targeted at a handful of business needs: customer services, contract services and order services. Each of these services has its own endpoint with a few methods to support working with each of the primary business types identified.

To be specific, a service could be developed to support working with customer data that would contain methods such as CreateCustomer, SearchCustomers, and GetCustomer. Each of these methods would either accept or return customer objects and if you retrieved a customer object using the GetCustomer service you could inspect the contracts that the customer has as well as the orders placed under each contract. Chances are that if you retrieved a customer using the GetCustomer method you would not be getting the populated contract objects along with it at that time. You would need to make additional calls to retrieve individual contracts and associated orders if that was the information you were after. The same concept should hold for working with contracts or orders. As you can see, the object hierarchy is maintained.

Assuming that this is the approach taken and the GetCustomer method returns a serialized object of type Customer once that customer object is deserialized in the client application it is easy enough to create contract objects, attach them to the contracts collection on the customer and create order objects and attach them to the contract objects and we have the same object oriented goodness on the client as we have on the server.

This is a standard approach when first building service enabled applications. Unfortunately this does not work well for service-oriented applications that are intended to be reused throughout the enterprise for other purposes beyond the initial application. Here's why.

Service Referencing

Let's think about developing the client side portion of our application, whether it is a web, WinForms or WPF application does not matter. To begin, we create a user interface for dealing with all things related to customers. We can create new customers, modify existing ones and search for customers based on various criteria. Once we have the user interface defined we add a service reference, using Visual Studio, to our previously created service which in turn generates our classes and proxies. We instantiate objects, add data supplied by the user and submit these objects to our services. Pretty standard stuff. Next we move on to developing the portion of the user interface that deals with contracts. We follow the same pattern and things are working well until we get to a namespace collision. When we added a reference to the customer related service, Visual Studio generated code for the classes that make up the return types and the request types our service supports. This will include all of the serializable types in the customer object graph.

When we add a reference to the contract service, Visual Studio will generate the code for the classes that make up the return and request types that this service supports. This will also include all of the serializable types in the contract object graph. In other words we end up with generated code for all of the classes described above twice! This is because each reference generates the classes in a distinct namespace. Even if you use the same reference name Visual Studio will alter them slightly to make them distinct. For example if both the first and second service references are given the name "localhost", Visual Studio will append a "1" to the end of the first reference making the namespace begin with "localhost1". You now have two Customer classes, localhost.Customer and localhost1.Customer, as well as two of every class in the respective object graphs.

Now your code is duplicated and stored in different types. You cannot create an instance of a localhost.Customer object and assign it to a variable of type localhost1.Customer. Not only do you not get object compatibility but you also end up with a whole bunch of equivalent classes under different namespaces. There are a few commonly used ways to deal with this problem:

1. Add a reference to the assembly that contains your objects to your UI projects, and alter the service references to use that/those assemblies rather than generating the code.
2. Alter the generated code to remove the duplicates.
3. Alter the code generation process to reference the assemblies containing your data objects.
4. Develop mapping code to translate from one type to another.

There are challenges with all of these. The first and third options may not be possible if you don't have access to the assemblies, perhaps you are referencing services that you did not develop, and perhaps the objects contain code that shouldn't reside on the UI side of things such as database access logic. The second option is simply fraught with perils as the code will be updated and need re-altering after every reference update, and if this is in mid development there may be many updates. The fourth option adds extra work and who needs extra work?

The Service-oriented Approach

The correct approach is to avoid these problems completely when developing your services. Here's how.

A Customer service should know about customers and data directly related to a customer only. Your Customer service methods should return a slightly different object graph than the object graph defined above. You will still have the customer object and you will still have the addresses for that customer but that's it. Break the graph at the collection of contracts. When the client requires the contracts for a customer they need to submit a request to the Contract service

asking for the contracts for a particular customer. This should actually be the same approach regardless of the object graph in use.

Whether the customer explicitly asks for the contracts or the system hides the implementation details and makes the call to retrieve contracts are simply implementation details. The fact that the object graph does not contain direct links to the contracts would not impact the user experience.

Let me explain. Regardless of the size of the object graph in use it should be a rare case in which the entire graph is populated and returned by a service call. Generally you will find that only a portion of the objects are in use for any given user action. To minimize the amount of data that is sent over a network only the relevant objects should be populated and returned. The code that is developed on the client side takes the responsibility for calling the appropriate services to populate further objects in the graph as the user requests them - this is often termed "lazy loading".

The goal of "lazy loading" is to retrieve only the data that is required so as to improve performance of the application. In fact, the simplified object graph better supports this design approach than the more complex graph does. With the complex graph, if you have a Contract object and need to perform some work with the related Customer for that contract you still need a sparsely populated Customer object that contains, at a minimum, the customer Id that can be used to retrieve the full customer object from the service.

With the simplified graph the Contract class contains a customer Id directly. The relations between the objects are still maintained, however with the simplified version the relationships are more explicit than implicit, as they are with the complex version. To bring this back to the user experience, the user should not know whether the underlying code has been developed using the simplified or complex graph. They should be able to navigate from a contract to the related customer just as easily. It is up to you, the developer, to make this experience seamless.

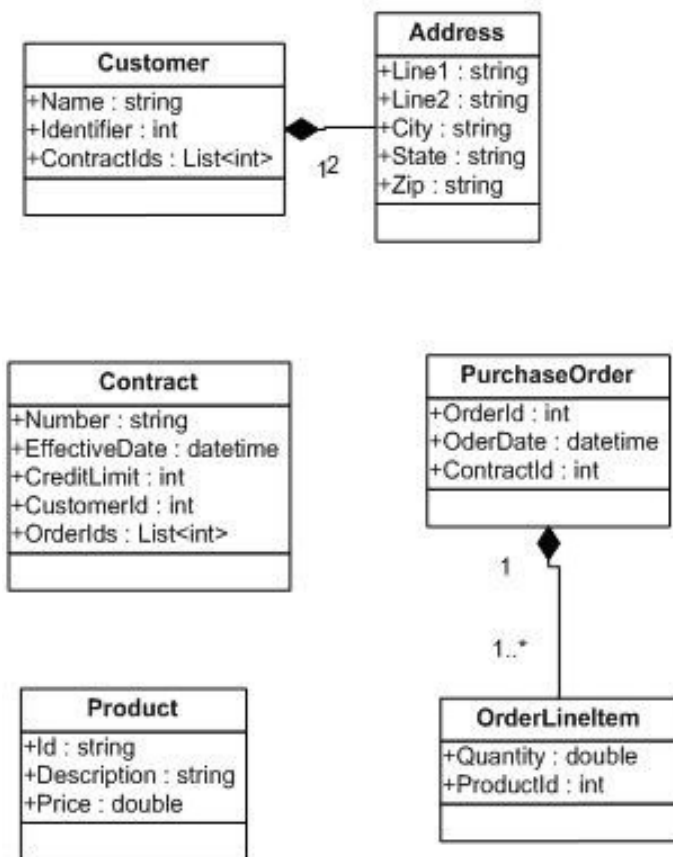


Figure 2

Again the object graph is kept simple as shown in Figure 2. As with the customer, contracts would not maintain a full customer object reference as part of the contract definition on the client side of the equation. A Contract object passed from the service would consist of only itself, no customer, and no PurchaseOrders. In place of the customer reference each contract object would maintain the customer identifier so as to be able to uniquely identify which customer the

contract belongs to and provide the means to “navigate” to the customer object when required.

As a side note, when requesting the collection of contracts for any given customer the collection of data returned should only consist of enough data in each object to uniquely identify the contract being sought, whether these objects are sparsely populated contract objects or a “light” version of the contract class does not really matter. You can then query for the full contract object based on the unique identifier that would be part of the collection of query results from the first request that returned the collection. Again, these are implementation details that are hidden from the user experience and need to be determined based on application needs.

The End Result

The end results of using this approach are:

- Returned result data is kept to the relevant bits. Even though this goal can still be achieved with a more complex object graph this approach enforces it.
- When adding a service reference using tools like Visual Studio the generated classes have a smaller object graph which avoids having multiple identical classes in different namespaces.
- Cleaner separation of duties.

Once you have adjusted to using this approach you will find that you need to create a mapping layer just behind the services - to map to and from the interfaces exposed by the services and your more complex object graphs. Either that or you can use simpler object graphs on the server side. However, these simple object graphs may not provide the functionality needed by the consumer of the service, especially if the consumer is a rich client application. In that case, a more complex object model can be designed and mapped to the simpler objects returned by the service. This keeps the service architecture simple while allowing the ability to use all of the OO principles that we’ve learned over the years. When it’s not needed, the simpler objects can simply be used as is with no additional work required.

By keeping the server side object graphs at the same simplicity as the service interfaces you will find that your code becomes cleaner, more modular, your classes become more cohesive and there is less coupling between your objects. In addition, the flexibility to reuse the customer service with other services which rely on customer information, but are sourced from a different repository, is easier to achieve. Merging multiple customer data systems into a single customer service also becomes much easier if the customer data is de-coupled from any other data. This now leads to a potential increase in service-orientation reuse, and a happier enterprise.

Conclusion

Both object oriented and service-oriented design and develop techniques have their place in modern systems development. Object oriented systems fit well in a stateful environment while a service-oriented approach requires a stateless environment. There is nothing wrong with the strong object oriented approach as described at the start of this paper however it will not serve you well if you try and expose those object graphs through a service.

With years of OO experience it’s easy to fall into OO design by default, but when designing systems we need to shift our mindset and think about what we are designing for. If it’s an SOA system, a traditional OO approach may not be the best. The tight coupling will get you in trouble as you expand the reach and reuse of your services throughout your enterprise. Keep the interfaces into your services simple and focused and you will find that your services become much easier to manage and become much more scalable.

To summarize, OO is, by its nature, stateful while SOA is, by its nature, stateless. This is where the impedance mismatch shows itself.

