Design Patterns for Domain Services

Fundamental Design Solutions for Service Oriented Architectures

Rob Daigneau
Rob.Daigneau@ArcSage.com

Abstract

Service Oriented Architecture (SOA) means many things to many people. For some it is a way to expose application capabilities to different types of consumers usually via platform-independent mechanisms. For others, it provides a new way to integrate systems or direct the manner in which they collaborate. Unfortunately, these all-encompassing definitions have left many wondering what SOA really is.

To date we have witnessed the rise of two general categories of services which I'll refer to as being **Enterprise Services** and **Domain Services**. Whereas the former are essentially composite services that typically leverage technologies such as Message-Oriented-Middleware, the latter are the building blocks upon which the composites depend. Each service category encompasses a distinct set of design solutions and is therefore worthy of individual attention.

The patterns that follow are early excerpts from a book forthcoming from Addison Wesley on this subject. This book will focus upon patterns specific to the creation of Domain Services. I'll expand upon the **Service Layer** concept [Stafford, Patterns of Enterprise Application Architecture], and will also show how Domain Services might be used with **Enterprise Integration Patterns** [Hohpe, Woolf].

Pattern Overview

Asynchronous Response Pull	How can a client submit a request to a long-running service operation and acquire a response, yet be free to move on to other work?
Dataset Element	How can service operations be designed to not only promote extensibility, but to also minimize the burden of having to maintain the signatures of each operation that uses the same data?
Command Driven Operation	How can the number of operations on a service be minimized so that the potential for high coupling with clients can be averted?
Dataset Batch	How can service operations be designed to help mitigate the inherent performance weaknesses of distributed computing?

Asynchronous Response Pull

Service Design Style: SOAP, REST

How can a client submit a request to a long-running service operation and acquire a response, yet be free to move on to other work?

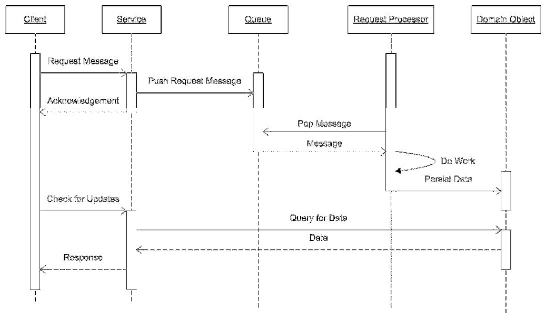
Some of the work incurred by client requests can be rather lengthy or "long-running" in nature. Of course, the definition of what this means depends upon the context of the problem, system requirements, and your perspective. In many cases, long-running work may be defined as requests that exceed a few seconds (e.g. the default time-out span of client proxies). Lengthy operations may also involve elaborate **Messaging Conversations** that transpire between several parties over many days. In the first situation, it is usually better not to block the client from moving on to other work. In the latter case, it's just not feasible to have the client waiting for the response.

While some operations can be tuned in order to make them run faster, there will be times when no amount of code optimization, refactoring, or database tuning can save you. The business use-cases for which these operations were written may just be terribly complex. The multi-party messaging conversation mentioned above is just one example. More commonly, complex operations may involve dozens of domain objects that each need to perform many queries or updates. We might try to look for ways to shorten the overall duration of the operation by having it invoke methods on the domain objects in an asynchronous fashion, but this isn't always feasible. On many occasions the results of one method must feed into the next method, and the same might be true for each successive call mediated by the service operation. So we're stuck with an operation that takes a long time to complete! But it gets even worse. If the operation simply processes each request as soon as it comes in, a tidal wave of requests could bring the systems behind the service façade to their knees.

When faced with this situation, service and client developers alike may concede that the client does not need to be blocked waiting for the service response. There are a few ways to handle this situation without having to make drastic changes to the service operation. The client may create a **Proxy** [GOF] that contains methods allowing for asynchronous invocation of service operations (e.g. **Polling Methods** and **Callbacks**). Unfortunately, clients that use these patterns remain vulnerable to timeouts, and responses are lost if the client crashes. Furthermore, these patterns do nothing to alleviate the risk of the service being overwhelmed by requests.

Another way to solve this problem is for the client to use **Request/Acknowledge** in conjunction with **Notifications**, a pattern that may be referred to as the **Message Relay** pattern. In this approach, the client submits a request message to a service, the service hands that request off to an asynchronous worker, and then returns a simple acknowledgement indicating that the request has been received. When the asynchronous worker completes, it sends a notification message (i.e. a type of **One-Way Message**) to a **Return Address** [EIP] provided in the original request. This notification message contains the final response. Unfortunately, the Message Relay pattern cannot be used in all situations. In order to implement that pattern, the client organization must create a **Service Endpoint** used to receive the notifications. The client may not want to alter the firewall rules of their network to allow inbound traffic, or they may not want to create or maintain services for some other reason. When service designers have a long-running operation, how can they possibly appease clients such as these?

Design service operations such that the submission of a request and the receipt of the ultimate response occur within separate client/service interactions. In the first exchange, the client sends a request to a service and receives a quick acknowledgement. The client will then issue subsequent requests to a different service operation or resource URI in order to pull back the response.



The **Asynchronous Response Pull** pattern is comprised of two separate Request/Response exchanges between a client and one or more services. The first interchange uses the **Request/Acknowledge** variation of the Request/Response pattern. In this exchange the service receives the request and generates a unique identifier called a **Request ID**[1]. This ID may be used by the client in subsequent requests when querying upon the status of the original request. Perhaps it should go without saying, but the client and service must agree upon both the data type used for the Request ID and the location where it will appear within the message. Once the Request ID has been generated, the service will add it to the message. Usually the best place to put this type of information will be in the **Message Header** because such information is not truly a part of the domain data found within the body of the message. After the Request ID has been generated and added to the message, the service will quickly push the message to a queue, or it may write the data from the message to a table[2]. Other than adding the Request ID to the message, the service does not act on the message at all. Once the message has been sent to a queue or written to a table, the service will quickly return an acknowledgement (i.e. a response message) containing the Request ID to the client.

Meanwhile, an asynchronous worker (a.k.a. **Request Processor**) will pop messages off of the queue or read new rows from the designated database table. In some cases it will process the request by instantiating an object in the **Domain Layer** [POEAA] in order to invoke the necessary domain logic. The request processor may also create a specific **Command** object [GO4] that encapsulates the process logic associated with the incoming message. In the most sophisticated scenarios, the request processor may be a workflow or rules engine. The possible implementation approaches for the request processor are many, and while this entity plays an important role in this pattern, its implementation is not central to this pattern. What is important here is that the service receives a request and quickly passes it on to some type of request processor that runs asynchronously in a process apart from the service.

Once the service has returned an acknowledgement, the client must retrieve the Request ID provided in that acknowledgement. The client will use this ID in a second Request/Response exchange in order to query upon the status of the original work request. Clients of SOAP services are frequently coded ahead of time to invoke a second known service operation designed for this purpose. In a more

[1] The Request ID may also be referred to as being a Correlation Identifier [EIP] or Transaction ID.

[2] Given the choice between using queues and database tables, the former is superior in many cases where inter-process or inter-application integration is required. For more information on this topic, I recommend *Enterprise Integration Patterns* [EIP]

sophisticated approach available to clients of both RESTful and SOAP services, the service may return a URI within the acknowledgement. The client may use this URI to dynamically construct a proxy so that it can check on the status of the initial request. While there is currently no common construct for RESTful services to pass this type of information back to clients, SOAP services may leverage the **ReplyTo** construct of **WS-Addressing**.

The client may execute the second request query at any time. It may even check on the status of the work request multiple times using the Request ID provided in the acknowledgement. Depending upon the design of the systems behind the service façade, inclusive of all request processors, the status of the work request may change with each call. For example, a request may return a status of *Open* on the first call, appear as *InReview* in the next call, and change to be *Fulfilled* in a final query by the client. Furthermore, the data provided by the service in each response may change significantly over time as well.

One characteristic of this pattern is that the client that invokes each subsequent request doesn't have to run on the same thread of execution as the original requestor. For example, a user may submit a request through one application, and when that application receives the acknowledgement it might save the Request ID to a database table. Another user working a different shift might use a different application to review work in progress, and by doing so, could initiate additional requests that check for updates using the saved Request IDs. Since responses can be retrieved by different threads or applications, this pattern also helps to provide resiliency upon the event of a crash in the initial requestor. Additionally, **Fault Messages** can be returned to the client during any interaction with a SOAP service. With RESTful services, error information may be passed back in the HTTP return codes.

The Asynchronous Response Pull pattern can be beneficial in other scenarios as well. First, the client can choose when to check for the response, an option that may be preferred if the client wants to be in charge. If the data in the response changes frequently over time, the client can repeatedly check in to discover the latest status of their request. While the Message Relay pattern puts the burden on the service to notify clients about prepared responses, the burden shifts to the clients when using this pattern. One disadvantage with this approach occurs when the client doesn't check back in a timely manner. When this happens there may be a significant delay between the time the response has been prepared and the time the client retrieves the response. If the client must be alerted as to the completion of their requests without delay, it may be better to use **Notifications** or **Callback Methods**.

This pattern provides organizations an effective means to throttle incoming requests, thereby protecting the system's resources from being overwhelmed. Organizations can easily "scale out" the servers hosting the services, yet throttle the requests by forcing them through a queue. At the other end of the queue one might find **Competing Consumers** [EIP] which can also be scaled out horizontally. The Asynchronous Response Pull pattern is therefore a viable choice when high loads are anticipated.

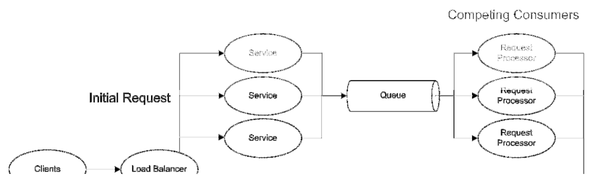


Figure 2: Asynchronous Response Pull can be scaled out by deploying the services to a load-balanced server farm.

This pattern incurs more network round-trips than the simple Synchronous Request/Response style of client/service interaction. The burden is placed upon the client to check back with the service in a timely manner, and if the client doesn't do this then there may be a significant delay between the time the response has been prepared and the time the client retrieves the response. Therefore, the client developer must put some thought into determining how frequently they should poll the service for a response. This frequency will vary for different applications.

Example: SOAP Service
This example shows a service that provides two operations, PlaceOrder and GetOrder. The first is used by clients to make the initial request, and the second is used to pull the response.

```
@WebService()
public class OrderService {

  @WebMethod(operationName = "PlaceOrder")
  public OrderStatus PlaceOrder
    ( @WebParam(name = "order")
      Order order)
  {
    OrderStatus status = new OrderStatus();

    String requestId = System.currentTimeMillis().toString() +
                     java.util.UUID.randomUUID().toString();

    status.setOrderStage("Open");
    status.setRequestId(requestId);

    order.setCorrelationId(requestId);

    FulfillmentSystem.SubmitOrder(order);

    return status;
  }

  @WebMethod(operationName = "GetOrder")
  public OrderStatus GetOrder
    (@WebParam(name = "requestId")
      String requestId)
  {
    return FulfillmentSystem.CheckOrderStatus(requestId);
}
```

This listing demonstrates how a unique Request ID may be generated by using simple Java APIs. The identifier is assigned to an *OrderStatus* object and an *Order* object that has been deserialized from the request message. This is used by the client to retrieve the response for the original request regardless of the keys used by internal systems to refer to the order.

Once these activities have been performed, the service proceeds to push the order to the fulfillment system by calling a static method named *SubmitOrder*. This method presumably serializes the order onto a JMS queue[3] so that it may be picked up by a request processor within the fulfillment system.

All of the logic which occurs before the acknowledgement is sent to the client must be designed with the highest levels of performance in mind. Therefore, it is best if the service does nothing more than attach the Request ID to the message, pass the message off to a request processor, and return an acknowledgment.

I decided not to show how clients would interact with this service as it should be rather obvious. Suffice it to say that after the client has submitted a request to PlaceOrder, it must retrieve the Request ID from the acknowledgement and then use it when calling the GetOrder operation.

Example: RESTful Service

This example shows how the Asynchronous Response Pull pattern might be used with a RESTful service. You will notice that the **Resource Mapper** pattern is used to map HTTP PUT and GET requests into operations named PlaceOrder and GetOrder, respectively.

```
@Path("/orders")
public class OrdersResourceMapper {

  private static String BaseURL =
                "http://www.acmeCorp.com/orders/";

  public OrdersResourceMapper() {;}

  @PUT
  @ConsumeMime("application/xml")
  @ProduceMime("text/plain")
  public String PlaceOrder(Order order) {

    String requestId = System.currentTimeMillis().toString() +
                     java.util.UUID.randomUUID().toString();

    order.setCorrelationId(requestId);

    FulfillmentSystem.SubmitOrder(order);

    return BaseURL + requestId;
  }

  @GET
  @Path("/{requestId}")
  @ProduceMime("application/xml")
  public Order GetOrder(
    @UriParam("requestId")
        String requestId){

    return FulfillmentSystem.CheckOrderStatus(requestId);
  }
}
```

[3] JMS, or *Java Message Service*, provides an API for a *Message Oriented Middleware* that supports the ability to send messages asynchronously between clients and servers. The .Net equivalent is MSMQ, or *Microsoft Message Queuing*.

Assuming that clients which issue PUT requests to this resource prepare requests in accordance with the structure defined by the Order class and also send these requests to a URI which maps to "BaseURL + /orders", the PlaceOrder method will receive each request and function in a manner similar to what was demonstrated in the SOAP Service example. The primary difference is that this version of PlaceOrder returns a String response (i.e. acknowledgement) which provides each client a URI it may use to inquire about the status of its order. The client may then issue a GET request to this URI to retrieve the order status.

It should be noted that, in each of these examples, clients are not prevented from inquiring about the status of orders submitted by other clients. Therefore, it should be apparent that one should also use appropriate authentication mechanisms such as **Identity Tokens** or **Digital Certificates**.

Related Patterns and Known Uses:

1. Brown, Kyle. *Asynchronous Queries in J2EE.*

   http://www.javaranch.com/journal/2004/03/AsynchronousProcessingFromServlets.html

   This article illustrates an approach that is quite similar to the pattern described here. The solution shows how long-running queries may be processed by a pair of servlets, a pair of JMS queues, and a Message-Driven Bean. The servlet that receives requests extracts pertinent information from the request, creates a Correlation ID [EIP], and then creates a Command object [GOF] which encapsulates the logic required to process the request. The command is provided the Correlation ID along with the data from the request message, and is then serialized by the servlet onto a designated JMS "Request Queue". Once this is done, the servlet returns the Correlation ID to the client.

   The Message-Driven Bean (MDB) which listens to the JMS request queue deserializes the command and invokes its logic through a well-known interface. Once the command has completed, the MDB serializes the response onto a special JMS "Reply Queue". A second servlet is used to dequeue the response that contains the Correlation ID [EIP] for the original request.

2. Snell, James. *Asynchronous Web Service Operations using JMS*

   http://www-128.ibm.com/developerworks/library/ws-tip-altdesign1/

   This article demonstrates a design approach nearly identical to the one shown in Kyle's article. While Kyle's article shows how to implement the pattern with servlets, this article shows how to do the same thing with web services.

3. WS-Polling

   http://www.w3.org/Submission/ws-polling/

   This specification defines a set of common SOAP extensions that may be used by clients to asynchronously retrieve messages stored by a service or a designated third party. The rationale for this specification was to free clients from the need of having to set up a service endpoint where services would return notifications (i.e. One-Way response messages containing the final results of a request). By allowing the client to poll for the response rather than having the client set up an endpoint, the client doesn't need to alter any firewall rules.

4. WS-Addressing

   http://www.w3.org/Submission/ws-addressing/

   This specification defines a way for any messaging participant to include address related information for services, service endpoints, and messages within a SOAP header.

References:

[EIP]

Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003

[GO4]

Gamma, Helm, Johnson, Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[POEAA]

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002

[REST]

Fielding, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Dataset Element

A service operation accepts parameters that are logically related and might be used in other operations.

Service Design Style: SOAP

How can service operations be designed to not only promote extensibility, but to also minimize the burden of having to maintain the signatures of each operation that uses the same data?

Developers frequently design service operations in such a way that they possess long input parameter lists. In so doing, they have inadvertently increased the coupling between clients and those operations, and have also set the stage for maintenance headaches. Service operations with this kind of "Flat API" are inherently inflexible. If ever the need arises to add or remove optional parameters, there is no viable way to do so within the operation's signature. Consider the WSDL provided in Listing 1.

Listing 1

Service operations that contain long input parameter lists are a common occurrence.

```
<portType name="HotelPortal">
  <operation name="CheckHotelAvailability">
    <input message="tns:CheckHotelAvailability" />
    <output message="tns:CheckHotelAvailabilityResponse" />
  </operation>
</portType>

<message name="CheckHotelAvailability">
  <part name="parameters" element="tns:CheckHotelAvailability" />
</message>

<message name="CheckHotelAvailabilityResponse">
  <part name="parameters"
         element="tns:CheckHotelAvailabilityResponse" />
</message>

<xs:complexType name="CheckHotelAvailability">
  <xs:sequence>
    <xs:element name="GuestCount"         type="xs:int" />
    <xs:element name="HotelChainCode"     type="xs:string" />
    <xs:element name="City"               type="xs:string" />
    <xs:element name="StateOrRegion"      type="xs:string" />
    <xs:element name="ArrivalMonth"       type="xs:int" />
    <xs:element name="ArrivalDay"         type="xs:int" />
    <xs:element name="ArrivalYear"        type="xs:int" />
    <xs:element name="DepartureMonth"     type="xs:int" />
    <xs:element name="DepartureDay"       type="xs:int" />
    <xs:element name="DepartureYear"       type="xs:int" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="CheckHotelAvailabilityResponse">
  <xs:sequence>
    <!-- element definitions would appear here -->
  </xs:sequence>
</xs:complexType>
```

The corresponding Java code for this WSDL is provided in Listing 2.

Listing 2

A Java web method with a "Flat API"

```
@WebMethod(operationName = "CheckHotelAvailability")
public TravelOptions CheckHotelAvailability(
  @WebParam(name = "GuestCount")          int    GuestCount,
  @WebParam(name = "HotelChainCode")      String HotelChainCode,
  @WebParam(name = "City")                String City,
  @WebParam(name = "StateOrRegion")       String StateOrRegion,
  @WebParam(name = "ArrivalMonth")        int    ArrivalMonth,
  @WebParam(name = "ArrivalDay")          int    ArrivalDay,
  @WebParam(name = "ArrivalYear")         int    ArrivalYear,
  @WebParam(name = "DepartureMonth")      int    DepartureMonth,
  @WebParam(name = "DepartureDay")        int    DepartureDay,
  @WebParam(name = "DepartureYear")       int    DepartureYear
 )
  throws InvalidDataRequestFault
{

  // implementation would appear here
}
```

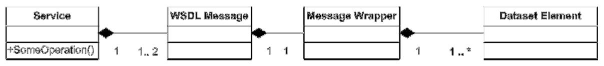
This seems innocent enough, but what would you do if you needed to add optional parameters to meet the requirements of new clients? Perhaps the client might want to indicate the guest's room preferences (e.g. room size, smoking or non-smoking, bed type, or room view)? Sure, we could insert these as additional parameters to the end of this list, and most client proxies wouldn't need to be updated because the majority of service frameworks are able to ignore parameters they don't recognize when these parameters occur at the end of the list. However, this is a pretty messy solution because now we're separating related parameters. It would be much nicer if we could insert these new parameters alongside the GuestCount so that we could keep all things "guest-related" together. Unfortunately, if we did try to squeeze new parameters into the middle of the argument list, we would likely incur a breaking change that would also raise several vexing questions. Should the service designer create a new operation, retire the old operation, and coax his clients onto the new one? Should he create a new operation and keep the older operation to maintain backward compatibility? Neither of these options seems very appealing.

If we anticipated the need to add new guest-related parameters, we might decide to move the GuestCount to the end of this operation so that all new parameters related to the guest would follow. However, this shuffling of parameters would do little to alleviate our problems in the long term because the same situation could arise time and again for other parts of the operation's signature. For example, we might want to further constrain the hotel search to those of a particular rating (e.g. 3, 4, or 5-star hotels), or to hotels that support certain discount programs (e.g. American Automobile Association discounts). The scenarios are endless.

Given all of this, it should be evident that "Flat APIs" not only produce brittle and inflexible service contracts, they also frequently cause developers to do a lot of repetitive coding. With respect to the guest-related parameters discussed above, the chances are good that these same arguments might be required in other service operations (e.g. an operation might later be created to maintain guest profiles). We can quickly conclude that this design style results in a maintenance nightmare for services and clients alike.

Given the obvious drawbacks of "Flat APIs", why do we continue to design service operations in this way? There are many possible reasons. Some developers may have a distorted understanding of what YAGNI[4] really means, and as a result, they haven't pondered the consequences of their short-sightedness. These developers may think that they can simply refactor their operations, but they do not take into account how clients will immediately become coupled to the service design once it is made public. In most cases, however, this style of design may have become a habit that was nurtured. Many of us were taught that **Simple Data Coupling** [Code Complete], wherein all of the parameters of a procedure are passed in as primitive non-structured data, was the best way to reduce coupling between modules. Many vendor tutorials and books have further enforced this philosophy by providing lessons that show service operations with "Flat APIs". While simple data coupling was recommended for procedural programming, and perhaps to a lesser degree for object-oriented programming, it tends to defeat extensibility and maintainability in SOA. But what recourse do we have, and how do we change?

Design all operations such that the input parameter list employs reusable data structures that group logically related sets of data together.



Dataset Elements (DSEs)[5] are reusable compound types that contain related data. They are roughly analogous to C Structs or Pascal Records. DSEs may be created in code as classes with XML serialization annotations, or they may be defined with XML Schema Language as Complex Types. The data within these structures may either be primitive data types (e.g. integer, string, etc.) or compound types. The child elements within a DSE may be marked to indicate that they are required or optional, their allowed values can be constrained, and the order in which they are serialized to XML within SOAP messages can be easily controlled. When a DSE contains one or many children, the structure of the DSE resembles a hierarchical tree.

DSEs do not encapsulate business logic. If they do have any methods, then these methods are only used to set or get the values of private member variables. DSEs provide a container in which data may be transferred from clients to services or vice

[4] YAGNI: You Aren't Gonna Need It: The philosophy that developers should not implement features until they are actually required.

[5] Dataset Elements should not be confused with Microsoft's platform specific data access technology that manages an in-memory cache of table-like data structures.

versa. Given this characterization, one might rightfully conclude that they are the same as **Data Transfer Objects** (a.k.a. DTO) [POEAA]. However, the context and forces that might lead one to use these patterns are somewhat different, as is their technical implementation. Martin Fowler describes a DTO as being "*An object that carries data between processes in order to reduce the number of method calls*". While DSEs are also used to carry data between processes, in this case between a client and the service, their main purpose is not to minimize method calls, a problem that is of great concern with distributed objects. Instead, their primary function is to group related data so that maintenance of the service's contractual obligation with its clients can be simplified. Given this, I like to think of DSEs as being specialized implementations of DTOs for use in services.

One might also recognize a similarity between DSEs and **Document Messages** [EIP]. While Document Messages provide the means to transfer related data as comprehensive messages between applications, DSEs occur as child elements within Document Messages. DSEs therefore provide a strategy to chunk out the data in a Document Message into smaller, meaningful, reusable, and manageable structures. DSEs can also be used to build up to a **Canonical Data Model** [EIP] in a very pragmatic way. This latter pattern is used when there is a desire to define an application-independent model that represents an enterprise-view of data.

The name of a DSE identifies an abstract type or logical grouping of data for some problem domain. This name should usually not be used to indicate the operation that manipulates the data. This is where the differences between DSEs and Document Messages become apparent. Whereas the names of messages oftentimes describe what a service should do (e.g. GetLoanTerms), or what part of a message exchange the data occurs in (e.g. GetLoanTermsResponse), DSEs do nothing of the sort. Instead, DSEs simply provide a logical grouping of data (e.g. LoanTerms) that can be used in any number of message exchanges, operations, and services.

Generally, DSEs should be kept as small and as compact as possible, and should only include child DSEs if those children are used in most use-case scenarios where the DSE appears. There are few reasons for these recommendations. First, as the breadth or depth of a DSE increases, it can become increasingly cumbersome to work with. Additionally, it is likely that some of the members of a large DSE will not be populated, and the absence of data could have unintended semantic importance to certain clients. Not only that, but these empty DSE members will still be serialized causing the server to do more work and increasing the message payload size; the net result is a negative impact on performance. So the trick in determining the right sizing of a DSE is to find that combination of members that seem to belong together and are usually always populated by the service operations they are used in.

DSEs provide a point of extensibility that is superior to what can be done within a service operation's definition. When DSEs appear as arguments within the service operation, it's easy to add new optional elements to any DSE by using the **Contract Amendment**[6] pattern. Not only that, but one can still maintain the fidelity of the entire service operation, even if the DSE appears in the middle of the operation's argument list. This is possible because changes made to a DSE will not alter the WSDL port, binding, message, or operation definition. Instead, this variability is pushed down into an XML Complex Type. This means that backward compatibility with clients that only know the older versions of the service contracts will not be compromised, and these clients will not need to generate new

[6] The Contract Amendment pattern describes an approach that allows service designers to publish minor (non-breaking) changes to service contracts by explicitly defining the addition of new optional elements at the end of a DSE. This pattern stands in contrast to Extension Elements, which are more open-ended.

proxies. Of course, the more difficult part is ensuring that any new logic introduced behind the service façade does not corrupt the results expected by clients using older versions of the DSEs. New DSEs can also be created from base DSEs, much like classes in object-oriented languages such as C# and Java can be extended.

DSEs provide a few other benefits as well. For one, developers may leverage XPath in order to extract out from the message only those DSEs they are interested in. Once the required DSEs have been plucked out of the message, validation of the DSE's structure and content is quite simple.

DSEs need not be used if you only have a few input parameters on a particular operation and the combination of parameters is not repeated across operations. However, whenever you are able to recognize a logical grouping of data that has the potential for reuse, then designing your operations to use DSEs might save you some grief in the future.

The Relationship between Dataset Elements and Domain Objects

DSEs shouldn't be created by simply annotating classes in the domain model with JAX-WS or WCF attributes. One problem with this approach is that it can be very difficult to serialize XML messages from an object graph because of the fundamental structural differences between object graphs and XML Infosets. XML messages are essentially tree structures, while object graphs may be unbounded structures with circular references. If an XML serializer is given the task of serializing an object graph with circular references (e.g. child objects pointing to parents and vice versa), the serializer will usually throw an exception because it will not be able to find a terminating node. While there are ways to work around this, they tend to be rather kludgey.

Another problem with annotating classes in the domain model with XML serialization attributes is that it creates a very strong coupling between the domain model and the messages being exchanged. If you ever need to redesign or reorganize your domain model, your changes could inadvertently alter the XML types that are generated, and this would ripple straight out through the service contract. Therefore, in order to set the stage for the independent evolution of the domain model and the clients that use your services, DSEs should be created as distinct and separate elements within the **Service Layer** [POEAA], and service designers should strive to use these constructs as a means to insulate clients from the internal design of both the domain layer and any database objects (i.e. tables, views, stored procedures) that are accessed. Of course, the challenge with this approach is that it will increase the amount of work you will need to do. Not only will you need to generate classes from your WSDL or vice versa, you will also need to write logic to move data back and forth between the DSEs and the domain model. This is a tedious and non-trivial task which might call for the use of an **Entity Mapper**[7], and is one of the biggest drawbacks to using DSEs.

The data in a DSE is ultimately mapped to one or many domain objects or database tables. There need not be a one-to-one correlation between DSEs and entities within the domain layer or database. In fact, the structure or design of DSEs may be quite different from these things. The reason for this is that the content of DSEs will typically be driven by client use-cases, and the data employed in a use-case doesn't always map neatly to a domain model or database schema. Regardless, on many occasions the design of a DSE will mirror the design of classes in your domain layer or the tables in your database.

[7] The Entity Mapper pattern is similar to the Message Translator [EIP] pattern. The primary difference is that it is used to map from DSEs that have been deserialized by the framework onto Domain Entities and vice versa.

A Few Parting thoughts on the Dataset Element

It can take significant amounts of time and effort to devise reusable types. Inevitably, different operations will use the information in the DSEs in various ways. The path of least resistance is to design DSEs as "one-off solutions" for each operation or a small grouping of operations. Unfortunately, this may result in a large assemblage of DSEs that have small differences but are strikingly similar. The consequence of taking this path may be a failure to achieve the desired reuse. The more difficult path is when the team attempts to design a **Canonical Data Model** [EIP]. Unfortunately, this approach may not be very pragmatic. Service designers must therefore carefully evaluate the trade-offs with these two extremes.

Example: Dataset Elements in Java and JAX-WS

This example shows how we might refactor the code from Listing 1 to use DSEs. All getter and setter methods have been eliminated for the sake of brevity.

```
@WebMethod(operationName = "CheckAvailability")
public AvailabilityResults CheckAvailability
  (@WebParam(name = "request")
  HotelSearchCriteria request) {

  // implementation here
}

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "HotelSearchCriteria",
  propOrder = {"GuestCriteria", "HotelCriteria",
               "ArrivalDate",   "DepartDate"})
@XmlRootElement(name = "HotelSearchCriteria")
public class HotelSearchCriteria {

  @XmlElement(name="GuestCriteria",required=true)
  public GuestInfo GuestCriteria;

  @XmlElement(name="HotelCriteria",required=true)
  public HotelInfo HotelCriteria;

  @XmlElement(name="ArrivalDate",required=true)
  public TravelDate ArrivalDate;

  @XmlElement(name="DepartDate",required=true)
  public TravelDate DepartDate;
}

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "GuestInfo",
  propOrder = {"GuestCount"})
@XmlRootElement(name = "GuestInfo")
public class GuestInfo {

  @XmlElement(name="GuestCount",required=true)
  public int GuestCount;
}
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "HotelInfo",
  propOrder = {"ChainCode", "City", "StateOrRegion"})
@XmlRootElement(name = "HotelInfo")
public class HotelInfo {

  @XmlElement(name="ChainCode")
  public String ChainCode;

  @XmlElement(name="City")
  public String City;

  @XmlElement(name="StateOrRegion")
  public String StateOrRegion;
}

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "TravelDate",
  propOrder = {"Month","Day","Year"})
@XmlRootElement(name = "TravelDate")
public class TravelDate {

  @XmlElement(name="Month",required=true)
  public int Month;

  @XmlElement(name="Day",required=true)
  public int Day;

  @XmlElement(name="Year",required=true)
  public int Year;
}
```

This is obviously a fair amount of code. Fortunately, the tools in most IDEs make quick work of all of this. You can see in the listing above that the CheckHotelAvailability operation has been refactored to receive a single DSE of the type HotelSearchCriteria. By having one argument in the parameter list, the service designer has much more flexibility in that they can do things like adding optional child elements while still not breaking the service contract. Some might say that because the operation signature has changed from being one with many arguments to one with only a single argument that we have now achieved a “Document Messaging” style of interaction rather than an RPC style. The truth of the matter is that even if I left this operation with a “Flat API”, the service framework would have wrapped my parameters in a document message of its own because, by default, most frameworks implement the “Document/Literal/Wrapped” pattern. This wrapper can be seen easily enough in the next listing.

```
<portType name="HotelPortal">
  <operation name="CheckAvailability">
    <input message="tns:CheckAvailability" />
    <output message="tns:CheckAvailabilityResponse" />
  </operation>
</portType>

<message name="CheckAvailability">
  <part name="parameters" element="tns:CheckAvailability" />
</message>
```

```
<message name="CheckAvailabilityResponse">
  <part name="parameters"
          element="tns:CheckAvailabilityResponse" />
</message>

<xs:element name="AvailabilityResults"
             type="tns:AvailabilityResults" />
<xs:element name="CheckAvailability"
             type="tns:CheckAvailability" />
<xs:element name="CheckAvailabilityResponse"
             type="tns:CheckAvailabilityResponse" />
<xs:element name="GuestInfo" type="tns:GuestInfo" />
<xs:element name="HotelInfo" type="tns:HotelInfo" />
<xs:element name="HotelSearchCriteria" type="tns:HotelSearchCriteria"/>
<xs:element name="TravelDate" type="tns:TravelDate" />

<!-- the wrapper element for the request -->
<xs:complexType name="CheckAvailability">
  <xs:sequence>
    <xs:element name="request" type="tns:HotelSearchCriteria"
                  minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<!-- the wrapper element for the response -->
<xs:complexType name="CheckAvailabilityResponse">
  <xs:sequence>
    <xs:element name="return" type="tns:AvailabilityResults"
                 minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="HotelSearchCriteria">
  <xs:sequence>
    <xs:element name="GuestCriteria" type="tns:GuestInfo" />
    <xs:element name="HotelCriteria" type="tns:HotelInfo" />
    <xs:element name="ArrivalDate"   type="tns:TravelDate" />
    <xs:element name="DepartDate"    type="tns:TravelDate" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="GuestInfo">
  <xs:sequence>
    <xs:element name="GuestCount" type="xs:int" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="HotelInfo">
  <xs:sequence>
    <xs:element name="ChainCode"     type="xs:string" minOccurs="0" />
    <xs:element name="City"          type="xs:string" minOccurs="0" />
    <xs:element name="StateOrRegion" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TravelDate">
```

```
  <xs:sequence>
    <xs:element name="Month" type="xs:int" />
    <xs:element name="Day"   type="xs:int" />
    <xs:element name="Year"  type="xs:int" />
  </xs:sequence>
</xs:complexType>
```

Related Patterns and Known Uses:

1. WCF Data Contracts

   Microsoft describes their DataContract construct as "a formal agreement between a service and a client that abstractly describes the data to be exchanged … A data contract precisely defines, for each parameter or return type, what data is serialized (turned into XML) in order to be exchanged".

   http://msdn.microsoft.com/en-us/library/ms733127.aspx

2. Javax.xml.bind.annotation.XmlType and Javax.xml.bind.annotation.XmlRootElement

   These allow Java developers to map a class or enumerated type to an XML Schema Type or Element, respectively

   http://java.sun.com/javaee/5/docs/api/javax/xml/bind/annotation/XmlType.html

   http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html

References:

[Code Complete]

McConnell, Steve. *Code Complete*. Microsoft Press, 1993

[Complex Type Definitions]

http://www.w3.org/TR/xmlschema-1/#Complex_Type_Definitions

The official description of how XML Schema Language is used to create complex types.

[EIP]

Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003

[POEAA]

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002

[YAGNI]

An acronym for the phrase "You Aren't Going to Need It". This is a philosophy from the Extreme Programming school of thought that suggests developers should only implement things when they are actually required.

http://c2.com/xp/YouArentGonnaNeedIt.html

Command Driven Operation

A service provides several operations that receive and manipulate the same data.

Service Design Style: SOAP

How can the number of operations on a service be minimized so that the potential for high coupling with clients can be alleviated?

Services typically offer operations for a set of related use-cases. A product company might publish an initial version of a Customer service with operations such as CreateCustomer, ReadCustomer, UpdateCustomer, and DeleteCustomer. A financial services company might supply a MutualFundOrder service with operations used to Buy, Sell, or Exchange mutual fund positions. In both examples, each operation usually receives the same data, and will probably invoke methods on the same **Domain Objects** [POEAA] in order to complete their respective functions. As the story often goes, with each new use-case that is identified, a new service operation is added thereby increasing the service's **Surface Area**. Large surface areas on services are unfortunately the cause for a few less than desirable side-effects. First, the client will be faced with a confusing glut of operations to choose from. Second, the clients might just use those operations and become dependent upon them. This increased dependency, or coupling, can make it harder for services and clients to evolve over time. If, for example, the service owner ever wants to consolidate or retire operations, the designers' options may be limited if the operations are heavily used. Ironically, the service has become a victim of its own success.

As the number of operations on a service increases, service designers are required to maintain more **Command Messages** [EIP][8] as well. With SOAP services, we generally find a one-to-one correspondence between Command Messages and service operations. In the Customer service example outlined above we would have four Command Messages … CreateCustomer, ReadCustomer, UpdateCustomer, and DeleteCustomer. These messages would either be explicitly created by the developer when starting with WSDL, or they might be auto-generated by the service framework if the developer started in the code (re: **XML Document Binding**). Unfortunately, when the service designer creates one operation for each client request, she will usually have an equal number of messages to maintain as well. This can quickly get out of hand as the service supports more and more use-cases.

Prior to the advent of service oriented design, developers sometimes addressed a proliferation of methods by using **Control Data** [Code Complete]. In this style of design, multiple methods would be consolidated down into a single method, and the client would pass control data in one argument to tell the method what it should do. Listing 1 illustrates this approach on a WCF service.

[8] Hohpe and Woolf describe Command Messages as constructs that may be used to "reliably invoke a procedure in another application".

Listing 1

A poorly designed WCF service that uses unconstrained Control Data.

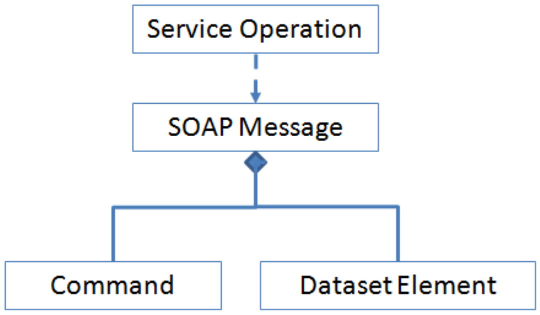
```
[ServiceContract]
interface ICustomerService
{
  [OperationContractAttribute(IsOneWay = true)]
  void RaiseCustomerEvent(string command, Customer customer);
}

public class CustomerService:ICustomerService
{
  public void RaiseCustomerEvent(string command, Customer customer)
  {
    switch (command)
    {
      case "Create":
        // call domain objects here
        break;
      case "Update":
        // call domain objects here
        break;
      case "Delete":
        // call domain objects here
        break;
      default:
        // throw a SOAP fault
    }
  }
}
```

The command parameter provides the client the means to tell the service whether it should create, update, or delete the data passed via the customer parameter. The problem here is that the consumer has to know something about the internal implementation of the service operation, specifically what values should be provided for the command parameter. If you ever decide to alter the internal implementation of the RaiseCustomerEvent operation, then you'll need to carefully coordinate this change so that your clients will know how to properly call this routine. Therefore, the deficiency with this approach is that the client becomes coupled to the internal implementation of the service.

Some also believe that this last approach is poor style because it complicates the operation and does little to describe what the operation actually does; it only suggests that it may do a number of things. In light of these very legitimate arguments, many suggest that we should create one operation for each logical command (e.g. CreateCustomer, UpdateCustomer, DeleteCustomer). At least with this approach, the purpose of the operation is explicit, and the implementation details of the service are hidden. Unfortunately, this puts us back at square one and we are left to battle a growing number of service operations that will inevitably multiply like rabbits. Is there a way that we can reduce the number of operations (and associated Command Messages) on a service, yet not suffer the negative consequences found in more traditional styles of programming?

Create a service operation that receives a common set of data and includes a parameter that may be set by the client to specify the nature of the request. Implement this parameter with enumerated types so that the client will not only understand how to use the operation, but will also be restricted to the choices within the enumeration.



Command Driven Operations seek to alleviate the potential for high degrees of coupling between clients and services by reducing the number of operations that receive and work with the same data. Rather than defining multiple operations that use the same **Dataset Elements**, the service designer creates a single operation that receives these elements. Since this pattern uses one request message (i.e. Command Message) for a variety of purposes, the service designer must introduce a mechanism that allows the client to indicate the reason why they are invoking the operation. This item, known in this pattern as a **Command Element**, is implemented as an enumerated type. Enumerated types are useful in this context because they constrain the set of allowable values the client may choose from. The values appearing in the enumerated type describe what the service operation will do should that particular value be selected by the client. So, while the name Command Element is reminiscent of the famous **Command** [GOF] pattern, it is not meant to encapsulate behavior or data related to a request. Instead, it is a simply a directive from the client.

Command Elements appear as children of **Command Messages** [EIP]. In most cases, the client will send a SOAP message that includes a single Command Element. This indicates that the service should use all of the Dataset Elements within the message to accomplish the task represented by the Command Element. In a sense, this pattern "takes the command out of the Command Message" by pushing the command instruction further down into the message body. What results is a SOAP message that is more generic. Listing 1 shows a typical SOAP envelope for a Command Driven Operation. In this example, the Command Message is named PlaceMutualFundOrder, and the Command Element has a value of "Buy".

Listing 1

A SOAP Message for a Command Driven Operation. Notice that the Command element indicates the nature of the client request.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" >
  <s:Header>
    <!-- Header detail would appear here -->
  </s:Header>
  <s:Body>
    <PlaceMutualFundOrder>
      <order xmlns:a="http://www.acmeCorp.org/schemas"
             xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <a:Command>Buy</a:Command>
        <a:BuySymbol>ACME</a:BuySymbol>
        <a:DollarValue>5000</a:DollarValue>
      </order>
    </PlaceMutualFundOrder>
  </s:Body>
</s:Envelope>
```

When a Command Driven Operation receives a message, it will use the Command Element to determine what the client would have the service do. Sometimes the operation will simply use this data to drive a simple conditional construct (e.g. switch or if statement). In more complex scenarios it might call upon a **Factory Method** [GOF] in order to acquire a related **Command** object [GOF]. In this latter case, the Command Driven Operation first acquires an interface to a family of Commands, and then invokes a method on that interface. The benefit of this approach is that the business logic associated with the Command Element remains fully encapsulated within specialized Command Objects, a design approach that usually facilitates the ease with which the business logic is maintained. The net result is that the Command Driven Operation merely dispatches the incoming requests to a Command, and that Command mediates all interactions with the **Domain Model** [POEAA].

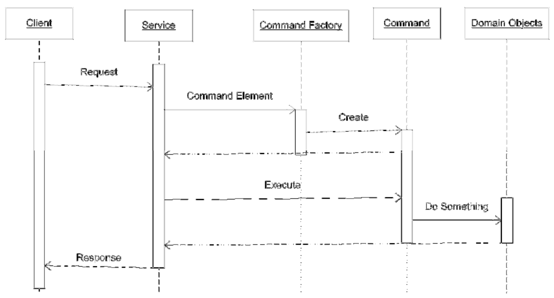


Figure 2: A Command Driven Operation passes a Command Element into a **Factory Method** [GOF]. The factory uses this information to decide what type of **Command** [GOF] to instantiate, and then returns a common interface for a family of commands. Next, the service invokes the Execute method on this interface, and by so doing, executes the business logic associated with the Command Element passed to the service.

This pattern facilitates the ease with which new "logical operations" may be introduced over time. I refer to these as being logical operations because this pattern can eliminate the need to implement concrete service operations on a given port. This technique goes a long way towards maintaining both forward and backward compatibility for clients because the WSDL port, binding, message, and operation definitions remain stable, and the variability (i.e. the addition or deletion of logical operations) is pushed down into the Command Element. You could, for example, gracefully remove commands from the Command Element enumeration. If a client were to send a request with a Command Element that is no longer supported, you could easily use a **Validating Interceptor**[9] to capture this, and then send the client a SOAP Fault informing them of this deprecation. Therefore, this pattern can be helpful when you anticipate that the "logical operations" for the service will change over time.

Command Driven Operations can sometimes resemble "God Methods". These are service operations that simply take on too much responsibility. However, judicious management of the enumerations included within the Command Element will mitigate this problem.

Some who consider using this pattern balk at the idea of defining Command Elements as enumerated types that are a part of the service contract. The typical objection is that if there's ever the need to add, change,

[9] Validating Interceptors make use of the Pipes and Filters [EIP] infrastructure found in popular service frameworks. They allow developers to create message validation logic that can be defined to execute before messages are delivered to the service implementation code.

or remove items in the enumeration, then clients must regenerate their **Service Proxies**. This is usually only true if the clients want to take advantage of new logical operations. Changes to the enums are rarely a problem. As for the deletion of specific items in the enumerated types, service designers can leverage Validating Interceptors in the manner described above.

Example: A Command Driven Operation developed in C# and WCF

The business scenario for this pattern is an overly simplified Mutual Fund service. Let's say that this service should offer clients the following logical operations …

- Buy a new fund (position[10]) with X dollars
- Sell all shares of an existing position
- Sell X shares of an existing position
- Sell a specific dollar amount of an existing position
- Sell all shares of an existing position in order to buy a new position
- Sell X shares of an existing position in order to buy a new position
- Sell a specific dollar amount in order to buy a new position

Upon first approaching this design problem, the designer might be tempted to create seven service operations, one for each use-case. However, upon further consideration, one can see that all use-cases involve the same data entities, albeit with slight variations to the input parameters that drive the logical operations.

Let's first review the WSDL and XSD for a service operation named PlaceMutualFundOrder. If you trace down through the WSDL, you'll see that this message contains a type named MutualFundOrder, which in turn contains an element of the type OrderCommand.

```
<wsdl:portType name="IMutualFund">
  <wsdl:operation name="PlaceMutualFundOrder">
    <wsdl:input name="PlaceMutualFundOrder"
                  message="tns:PlaceMutualFundOrder" />
    <wsdl:output name="PlaceMutualFundOrder"
                  message="tns:PlaceMutualFundOrder" />
  </wsdl:operation>
</wsdl:portType>

<wsdl:message name="PlaceMutualFundOrder">
  <wsdl:part name="Order" element="q1:Order"
             xmlns:q1="http://www.acmeCorp.org/schemas/" />
</wsdl:message>

<xs:element name="MutualFundOrder" nillable="true"
             type="tns:MutualFundOrder" />
<xs:element name="Order" nillable="true"
             type="tns:MutualFundOrder" />
<xs:element name="OrderCommand" nillable="true"
             type="tns:OrderCommand" />
```

[10] When you own shares of a stock or mutual fund, you have a *Position* in that stock or fund

```
<xs:complexType name="MutualFundOrder">
  <xs:sequence>
    <xs:element name="Command" type="q1:OrderCommand" />
    <xs:element name="BuySymbol" minOccurs="0" nillable="true"
                 type="xs:string" />
    <xs:element name="SellSymbol" minOccurs="0" nillable="true"
                 type="xs:string" />
    <xs:element name="Value" minOccurs="0" type="xs:decimal" />
    <xs:element name="ConfirmationId" minOccurs="0" nillable="true"
                 type="xs:string" />
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="OrderCommand">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Buy" />
    <xs:enumeration value="SellAllShares" />
    <xs:enumeration value="SellShares" />
    <xs:enumeration value="SellDollarAmount" />
    <xs:enumeration value="SellAllSharesToBuy" />
    <xs:enumeration value="SellSharesToBuy" />
    <xs:enumeration value="SellDollarAmountToBuy" />
  </xs:restriction>
</xs:simpleType>
```

Granted, looking at WSDL and XSD can give even the most seasoned developer a headache. Therefore, let's move right along to the platform code associated with the WSDL you see above. For the sake of brevity, public fields are used rather than properties in all classes.

```
[ServiceContract]
public interface IMutualFund{
  [OperationContract]
  PlaceMutualFundOrder PlaceMutualFundOrder(
                            PlaceMutualFundOrder order);
}

[MessageContract(
  WrapperNamespace = "http://www.acmeCorp.org/schemas/",
  IsWrapped=false)]
public class PlaceMutualFundOrder{
  [MessageBodyMember( Name="Order",
               Namespace="http://www.acmeCorp.org/schemas/")]
  public MutualFundOrder Order;
}

[DataContract(Name="MutualFundOrder",
  Namespace="http://www.acmeCorp.org/schemas/")]
public class MutualFundOrder{
  [DataMember(Order= 1, IsRequired= true)]
  public OrderCommand Command;

  [DataMember(Order= 2, IsRequired=false)]
  public string BuySymbol;
  [DataMember(Order= 3, IsRequired=false)]
  public string SellSymbol;

  [DataMember(Order= 4, IsRequired=false)]
  public decimal Value;

  [DataMember(Order= 5, IsRequired=false)]
  public string ConfirmationId;
}

[DataContract]
public enum OrderCommand{
  [EnumMember(Value="Buy")]
    Buy,
  [EnumMember(Value = "SellAllShares")]
    SellAllShares,
  [EnumMember(Value = "SellShares")]
    SellShares,
  [EnumMember(Value = "SellDollarAmount")]
    SellDollarAmount,
  [EnumMember(Value="SellAllSharesToBuy")]
    ExchangeAllShares,
  [EnumMember(Value = "SellSharesToBuy")]
    SellSharesToBuy,
  [EnumMember(Value = "SellDollarAmountToBuy")]
    SellDollarAmountToBuy
}

public abstract class MutualFundCommand{
  protected MutualFundCommand() { ;}
  public abstract string Execute();
  public abstract bool HasValidationErrors();
  public abstract List<string> ValidationErrors();
}

public class MutualFund : IMutualFund{
  public PlaceMutualFundOrder PlaceMutualFundOrder(
                             PlaceMutualFundOrder request){
    PlaceMutualFundOrder response = request;

    MutualFundCommand cmd =
                       CommandFactory.GetCommand(request.Order);

    response.Order.ConfirmationId = cmd.Execute();
    return response;
  }
}

public class CommandFactory{

  private static Dictionary<string, MutualFundCommand>
    _commands;

  public static MutualFundCommand GetCommand(
                                  MutualFundOrder order)
  {
    InitializeRegistry();
    MutualFundCommand command =
      ( _commands[order.Command.ToString()] ).Clone();

    command.setOrder(order);

    return command;

  }

  private static void InitializeRegistry()
  {
    if (_commands != null) return;

    // excluded thread-safe locking code to simplify a bit

    _commands =
        new Dictionary<string, MutualFundCommand>();

    AddPrototypicalInstancesOfCommands();
  }

  private static void AddPrototypicalInstancesOfCommands()
  {
    _commands.Add("Buy", new BuyCommand());

    _commands.Add("SellAllShares",
                            new SellAllSharesCommand());
    // add other commands here
  }
}
```

There is certainly much to take in with this example as it involves several patterns. At the top of the listing we see that IMutualFund defines an interface used for the WSDL port definition. The operation PlaceMutualFundOrder on this interface indicates that a **Command Message** [EIP] of the type PlaceMutualFundOrder is used as both the request and response message type. Note that a WCF MessageContract is used and “automatic wrapping” is turned off in order to control the name of the message. It only contains a single message part of the type MutualFundOrder so that it will remain compliant with the WS-I Basic Profile specification. MutualFundOrder is a **Dataset Element** which contains a number of parameters including a **Command Element** of the type OrderCommand. This enumeration defines the logical operations supported by the PlaceMutualFundOrder operation.

Next we see an abstract class named MutualFundCommand. This class is used to define a common interface for a family of **Commands** [GOF]. The service implementation class named MutualFund receives the PlaceMutualFundOrder Command Message and passes it into a **Parameterized Factory Method** [GOF] named GetCommand. This factory method uses a **Registry** [POEAA] that contains **Prototypes** [G04] of MutualFundCommands. It retrieves a prototypical instance of the desired command and clones that instance. Once the command has been cloned, the factory method provides the Order to the command via the setOrder method and then returns the command back to the PlaceMutualFundOrder operation. Now the service operation is able to call the Execute method in order to carry out the client’s request

Related Patterns and References:

[Code Complete]

McConnell, Steve. *Code Complete*. Microsoft Press, 1993

[EIP]

Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003

[GO4]

Gamma, Helm, Johnson, Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[POEAA]

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002

Dataset Batch

Service Design Style: SOAP

How can service operations be designed to help mitigate the inherent performance weaknesses of distributed computing?

In the not too distant past we discovered that our application of Distributed Object technologies (e.g. CORBA, DCOM) created significant performance problems. What should have been obvious to everyone is now accepted as conventional wisdom. This wisdom may be imparted through a simple metaphor. If the time required to execute a procedure in-process is akin to walking across the room, then the time used to execute a process on another machine is like travelling in a spaceship to *Proxima Centauri*[11]. This analogy is as true for SOA as it was for distributed object architectures. There will always be a performance penalty to pay for "out-of-process" calls, even when clients and services are deployed to the same machine[12]. When we decide to use SOA, we are selecting an architectural approach that has an inherent Achilles heel. This factor must be considered when designing services, but all too often it is not.

Service designers usually provide operations that are meant to complete singular and relatively small transactional requests. Examples include operations named UpdateCustomer and ExchangeFund[13]. The common characteristic shared by these operations is that each connotes an atomic transaction whereby the information extracted from the **Dataset Elements** (DSEs) are either saved in their entirety or are rolled back. In many cases, this style of design will suffice. However, clients oftentimes possess multiple datasets they would like to push over to the service. Sometimes these datasets will need to be persisted within the scope of a larger transaction, other times they won't. Regardless, the client is preparing for that proverbial trip across the galaxy.

[11] The closest star to Earth, excluding our own Sun of course.

[12] When clients and services are co-located, the performance penalty can be partially attributed to the time it takes to marshall and unmarshall the data across processes. It's not as bad as cross-machine calls, but it still hurts.

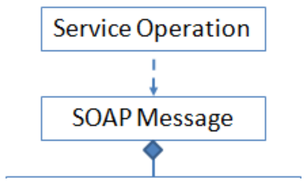
[13] Assume that this operation provides the ability to sell either a dollar or share amount in some mutual fund position in order to buy a position in another fund. In other words, the operation sells one fund to buy another.

The decision to design services to process "small" and singular entities one at a time causes a few problems for both the client and the service. First, the client must call the necessary service operations for each Dataset Element or logical business transaction repeatedly. If, for example, the client has five customers whose information needs to be updated, the client would have to call the UpdateCustomer operation five times. If we wanted to execute three ExchangeFund orders, we'd have to call that operation three times. Each invocation of a service operation incurs a network round-trip, so the act of updating five customers or executing three fund exchanges would incur a significant amount of network activity, which would also have a negative impact on overall performance.

The second problem that can arise with service operations that process "small" and singular entities relates to the issue of how service errors are detected and handled. All clients that call upon these types of services to save multiple datasets would also be responsible for the logic required to detect and handle any errors that might occur part way through the execution of the larger logical transaction. So, in our example, if a client has three ExchangeFund orders to execute, and the first two invocations complete successfully but the last one fails, the client would be responsible for implementing the exception logic (i.e. the error detection and rollback logic). Making the client responsible for these things is usually undesirable for a few reasons. First, a common goal of services is to encapsulate both the "successful execution path" of a transaction and the related exception handling logic. The design approach in question obviously violates this principle. Furthermore, consistency of any error handling logic implemented in the clients becomes much harder to achieve as the number of clients increase. In addition to all of this, the use of traditional error handling techniques (e.g. rolling data back) in distributed scenarios can be slow and unreliable, especially when transports like HTTP are used or when the client and service are developed for different platforms. It all adds up to many "inter-stellar sojourns".

While there are many benefits to be realized from SOA, we must be realistic and recognize its inherent weakness. Is there anything we might learn from the space-traveler analogy? Perhaps it is this … maybe we should load up the ship before we depart.

Design service operations to receive messages that contain collections of related Dataset Elements that could be processed within a single transaction mediated by the service.



Dataset Batches provide clients the means to submit messages containing collections of related DSEs. With this pattern, the client can send a single batch of DSEs and related commands to a service, thus minimizing network round trips. Services that use Dataset Batches can effectively manage each dataset within the scope of a larger service-side transaction. The service also encapsulates all error handling logic for the batch. All of this results in an encapsulated, consistent, and centralized approach for such matters, and tends to greatly simplify the clients.

The simplest form of this pattern exists when a given service operation carries out the same transaction for a collection of related DSEs sent by the client. For example, an operation named UpdateCustomers would receive a collection of Customer DSEs and would typically execute the appropriate domain layer logic repeatedly for each Customer DSE found in the collection. To do this, the service would leverage classes that implement well known domain layer patterns (e.g. **Table Module**, or **Domain Model** [POEAA]) or data source access patterns (e.g. **Table Data Gateway**, **Row Data Gateway**, **Active Record** [POEAA]). If any problems occurred along the way, then the logic provided through these same patterns could be used to effectively undo these updates. By sending a batch of datasets to a service, network traffic is minimized and consistency in error handling for all of the DSEs that should be processed within the transaction is ensured.

A more complex usage of this pattern occurs when a service operation is responsible for executing the logic associated with many different transaction types. In this case, the collection received by the service is still comprised of related DSEs, but now each contains a different **Command Element** (re: **Command Driven Operation**)[14]. After the

[14] In some variations of this pattern the client will send a collection of Command Elements, each of which contains a DSE.

service has received the client's request, it will iterate through the DSE collection and extract the Command Elements for each. These elements provide clients the means to tell the service what transaction should be invoked for the associated DSE. This usage of the Dataset Batch pattern provides clients the ability to submit a series of disparate requests within a single message. The client may even want the service operation to execute all of the commands within the scope of one logical transaction. For example, a client might send a batch of Customer DSEs which contain a mix of create, update, and delete commands to a service operation named SaveCustomers. This use of the pattern not only helps to minimize network traffic and ensure consistency in error handling, it also tends to reduce the service's **Surface Area**, which can help to minimize client/service coupling.

Use of this pattern does tend to increase the relative size of the message payloads when compared to the payloads of operations that handle "small" and singular entities or transactions. This being said, the performance penalty attributable to the somewhat larger message payload is usually offset by the performance gains achieved by minimizing network activity.

Another issue that sometimes crops up with this pattern is that clients which typically submit requests for a service to process a single DSE at a time may find this approach to be a bit annoying. These clients are forced to create messages which contain arrays of DSEs, even though the client might only submit one DSE in the collection. In an effort to pacify such clients, service designers will sometimes leave the "singular" versions of the operations available on the port. In order to minimize code duplication, the service implementation that uses the Dataset Batch may call upon the singular version of the operation to process each DSE passed in the collection. The disadvantage of this approach is that it introduces complexity on the service side, and also increases the service's Surface Area.

Finally, since the service is responsible for processing multiple DSEs within a single request, it has much more work to do than if it were to only process a single DSE. Consequentially, the response time of the operation may become excessive. Therefore, clients that use these types of service operations should consider using the Dataset Batch in conjunction with asynchronous invocation patterns (e.g. **Polling Method**, **Callbacks**). If a response is required by the client, then the service designer should consider also using patterns such as the **Asynchronous Response Pull** or the **Message Relay**. If a response is not required then the message containing the **Dataset Batch** may be sent to an **Event Sink**.

Example: A C# service receives a Dataset Batch

The first example shows how a WCF service operation named UpdateCustomers can be designed to receive a batch of Customer DSEs and save all information from these DSEs as part of one logical transaction.

```
[ServiceContract]
public interface ICustomerService{
  [OperationContract]
  [XmlSerializerFormat]
  CustomerMessage UpdateCustomers(CustomerMessage request);
}

[MessageContract(IsWrapped=false)]
public class CustomerMessage{
  [MessageBodyMember(Name = "ArrayOfCustomers")]
  public ArrayOfCustomers ArrayOfCustomers;
}

[Serializable]
[XmlType("ArrayOfCustomers")]
[XmlRoot("ArrayOfCustomers")]
public class ArrayOfCustomers{
  [XmlArray]
  public Customer[] Customers;
}

[Serializable]
[XmlRoot("Customer")]
[XmlType("Customer")]
public class Customer{
  [XmlAttribute("Id")]         public int Id;
  [XmlAttribute("FirstName")]  public string FirstName;
  [XmlAttribute("LastName")]   public string LastName;
  [XmlAttribute("Address")]    public string Address;
  [XmlAttribute("City")]       public string City;
  [XmlAttribute("State")]      public string State;
  [XmlAttribute("ZipCode")]    public string ZipCode;
  [XmlAttribute("UpdateDate")] public DateTime UpdateDate;
}

public class CustomerService : ICustomerService{
  public CustomerMessage UpdateCustomers(CustomerMessage request)
  {
    CustomerMessage response = request;

    using (TransactionScope txnScope = new TransactionScope())
    {
      try
      {
        foreach (Customer customer in
                        response.ArrayOfCustomers.Customers)
        {
         // Invoke logic in the business or data source access
         // layers in order to update customer information given
         // data provided in the current customer Dataset Element
        }

        txnScope.Complete();
      }
      catch (Exception ex)
      {
       // Perform any logic required to handle the exception.
       // All updates will automatically be rolled back.
      }
    }

    return response;
  }
}
```

The interface ICustomerService in this listing contains a single operation named UpdateCustomers that receives and returns a CustomerMessage[15]. This message contains one message part so that compliance with the **WS-I Basic Profile** specification will be maintained; this message part simply wraps an array of Customers.

When the UpdateCustomers method receives a request, it first directs the response message to reference the request message. This is done so that we can echo back to the client the information it sent. After this statement has completed, the service instantiates a TransactionScope. This is a .Net construct that is roughly equivalent to starting a database transaction. Next, we iterate through the Customers in the response object and invoke logic in either the business or data source layers in order to persist the information passed in via the Customer DSEs. If all updates complete without any exceptions, then the updates will be committed by calling the Complete method of the transaction scope object. If any exceptions did occur, then the catch block will be executed, and all updates will automatically be rolled back.

Example: A Java service receives a Dataset Batch

This example demonstrates how a Java service can be designed to process many different transaction types for the Dataset Elements provided. It does this by looking at the Command Element associated with each DSE.

```
@WebService(targetNamespace = "http://www.acmeCorp.org",
  name="MutualFundInterface")
public interface MutualFundInterface {

  @WebMethod(operationName = "PlaceMutualFundOrders")
  @WebResult(name="MutualFundOrdersMessage")
  public MutualFundOrdersMessage PlaceMutualFundOrders
    (@WebParam(name = "orderRequest")
     MutualFundOrdersMessage orderRequest);
}

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name="MutualFundOrdersMessage", propOrder = {"Orders"})
@XmlRootElement(name = "MutualFundOrderMessage",
           namespace="http://www.acmeCorp.org/schemas")
public class MutualFundOrdersMessage {
  @XmlElement(name="Orders",
               namespace="http://www.acmeCorp.org/schemas/")
  public List<MutualFundOrder> Orders;
}

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "MutualFundOrder",
propOrder = { "Command","BuySymbol","SellSymbol","Value",
                "ConfirmationId","ValidationErrors"})
@XmlRootElement(name = "MutualFundOrder",
          namespace="http://www.acmeCorp.org/schemas")
public class MutualFundOrder {

  @XmlElement(name="Command",required=true)
```

[15] Notice that the *OperationContract* for *UpdateCustomers* uses the *XML Serializer* rather than WCF's default *DataContract Serializer*. This was done so that types containing XML attributes can be properly serialized.

```
  public MutualFundOrderCommand Command;

  @XmlElement(name="BuySymbol",required=false)
  public String BuySymbol;

  @XmlElement(name="SellSymbol",required=false)
  public String SellSymbol;

  @XmlElement(name="Value",required=false)
  public java.math.BigDecimal Value;

  @XmlElement(name="ConfirmationId",required=false)
  public String ConfirmationId;
}

@XmlEnum(String.class)
public enum MutualFundOrderCommand {
  Buy,
  SellAllShares,
  SellShares,
  SellDollarAmount,
  ExchangeAllShares,
  SellSharesToBuy,
  SellDollarAmountToBuy
}

public abstract class  MutualFundCommand {
  public abstract String Execute();
  public abstract void Undo();

  private MutualFundOrder _order;

  public MutualFundCommand(){;}

  public MutualFundOrder getOrder() {
    return _order;
  }

  public void setOrder(MutualFundOrder order) {
    this._order = order;
  }
}

@WebService(
  endpointInterface="ServiceContracts.MutualFundInterface")
public class MutualFund {
  public MutualFundOrdersMessage PlaceMutualFundOrders
                                (MutualFundOrdersMessage request)
  {
    MutualFundOrdersMessage response = request;

    List<MutualFundCommand> commands =
                            new ArrayList<MutualFundCommand>();

    MutualFundCommand currentCommand;

    try{
      for( MutualFundOrder order : response.Orders){
        currentCommand = CommandFactory.GetCommand(order);
        commands.add( currentCommand  );
        order.ConfirmationId = currentCommand.Execute();
      }
    }
    catch(Exception ex){
      // perform appropriate logic to handle the
      // current exception here

      // Undo all commands
      for(MutualFundCommand command : commands){
        command.Undo();
      }
    }

    return response;
  }
}

public class CommandFactory {

  public static MutualFundCommand GetCommand(
                                       MutualFundOrder order){

    if(order.Command == MutualFundOrderCommand.Buy){
      return new BuyCommand(order);
    }

    if(order.Command == MutualFundOrderCommand.SellAllShares){
      return new SellCommand(order);
    }

    // etcetera

    return null;
  }
}

// A sample command object
public class BuyCommand extends MutualFundCommand {

  public BuyCommand(MutualFundOrder order){
    this.setOrder(order);
  }

  @Override
  public String Execute() {
    // command logic goes here;
    return "something";
  }

  @Override
  public void Undo() {
    // logic to undo this command goes here
  }
}
```

This listing starts by showing an interface used to define a service contract. This contract contains one operation named PlaceMutualFundOrders that receives and returns a message of the type MutualFundOrdersMessage. This type serves as a wrapper for a List of MutualFundOrder Dataset Elements. Each of these DSEs contains a Command Element of the type MutualFundOrderCommand. This element provides clients the means to direct the service to use each DSE for a particular transaction type (e.g. Buy, SellAllShares, etc.).

When the service receives the client's request, it first copies the request to a response message of the same type so that it can echo the client's request back. It then instantiates an array of MutualFundCommand objects. This abstract class serves as the base class for a family of **Command** Objects [GOF]. Each concrete specialization of a Command would contain the logic specific to one of the transaction types defined by the MutualFundOrderCommand enumeration. A skeleton example of one of these Commands is provided in the BuyCommand class.

After the service operation instantiates the array of MutualFundCommand objects, it iterates through the array of MutualFundOrder DSEs accessible via response.Orders. For each DSE retrieved from this list, it will call upon a **Factory Method** [GOF] of the type CommandFactory in order to acquire a concrete Command Object. It then adds the concrete command to the array of MutualFundCommand objects, and calls the Execute method of that object. Presumably, each concrete command returns a Confirmation ID, which the service operation then assigns to the current order. If any exceptions are trapped in the course of executing the commands, the service will iterate through the current array of MutualFundCommand objects and direct each one to Undo whatever it had done within its Execute method. The details of how the Execute and Undo methods might be implemented are beyond the scope of this book.

Related Patterns and References:

[GO4]

Gamma, Helm, Johnson, Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[POEAA]

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002

[WS-I Basic Profile]

Defines a set of specifications whose purpose is to promote interoperability.

http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

[XML Substitution Groups]

XML Schema Part I: Structures Second Edition

http://www.w3.org/TR/xmlschema-1/