1. Introduction:

This document describes the use case scenarios for the web services description language. The use case scenarios are intended to describe the ways how we see the web services being described and used in real life. The use case scenarios, since they describe the usage scenarios, in some cases may also have overlap with the use case scenarios of the web services architecture.

2 Scenarios:

UC0001 – [WS] Fire-and-forget: A metrics collection service exposes an interface for applications running in an environment to report their application usage metrics. Instead of reporting individual metrics, these applications report a computed sum that represents the summary of usage their usage. Therefore, the loss of a message is not important because the next update will again provide an updated summary. The target web service exposes an interface to report those metrics. In the interest of efficiency, the client applications are not interested in receiving any faults because and just want to send a message and forget about it until the next time.

UC0002 – [WS] One-way message with guaranteed delivery: A web service provides a messaging service. This web service is a higher level interface over enterprise messaging capabilities such as JMS. On the publisher side, it exposes an interface that allows applications to publish messages to a logical messaging channel. The publishing clients do not expect to receive a response to their invocation however it is important for them to know that the message has been delivered. So while they make a one-way request, they do want receive faults if the message could not be successfully published, whether due to a system error on the server side or due to an application error on the server side.

UC0003 – [WS] Multiple faults: A web service interface method can fail due to several reasons. The faults raised by the method may be semantically different from each other and further more, some of the faults may be standard faults defined for a group of web services. For example, in an accounting system, there may be a general "creation fault" defined for indicating the failure such as out of resources or PO already exists. The creation of PO could also fail because the data provided to initialize the PO is invalid. The web service method "createPO" might then fail because of any of the reasons described above and may want to raise separate faults depending on the reason for failure.

UC0004 – [WS] Service level attributes: Two web services, implementing the interface for "looking up for insurance providers", from different sources are offered in a registry. One of the two services actually performs extensive data validation on the data provided, for example making sure that the zip codes in the address provided are valid", while the other web service assumes that the data provided is valid and searches for insurance providers has already been validated and uses it to perform its search without any further validation. The interface was developed by an industry consortium that agreed to reflect

the data validation capability of the services as a service-level attribute. Some intelligent registries may then actually allow search criteria that can be predicated on these service-level attributes or alternatively, the client application may check the value of the service level attribute itself at runtime to find out its value. The service-level attribute may be mapped to accessor methods which can be invoked either by the intelligent registry as part of executing the search query or by the client application itself.

UC0005 – [WS] Operation-level attributes: In an advanced architecture where distributed transactions are supported, a web service may want to declare some of its operations as transactional as opposed to the entire interface being transactional. A web service offering various financial related web services may be able to verify a buyer's credit in a non-transactional manner but may require the client application to start a transaction before invoking the operation to prepare an invoice. The target web service may have a declarator on the method specification that indicates that the operation for invoicing requires transaction.

UC0006 – [WS] Document centric computing: A web service is ebXML based web service that requires document-oriented computing. The operations that its interface includes are all actually asynchronous messages with no parameters. All the data to the messages is sent as document attachments. Its description document will then describe the data expected by its operations in terms of the expected attachments and not in terms of its parameters.

UC0007 – [DW] Print Data Transformation: A service is available that will take data from a mobile device (PDA, Cell Phone) to be printed in a variety of formats (text, pdf, html) and will transform the data into a printable format (PostScript, PCL) and deliver it to a specified printer.

UC0008 – [DW] Document delivery: A service is available that will accept a document scanned from a multifunction device and deposit it in an archive. An e-mail or other notification will be sent to the specified recipient with a URL pointer to the document. The document can then be viewed, copied or printed by the recipient. End-to-end encryption will be available as well as recipient authentication.

UC0009 – [DW] Image manipulation: A service is available to take an image in a number of formats (JPG, PNG, TIFF, etc.) with potentially very large amounts of data (hundreds of megabytes) and apply transformations and other processes (de-speckle, de-skew, etc.) specified to the image and then returned the resultant processed image back to the requestor in the requested format.

UC0010 – [DW] Printer Maintenance Service: A service is available that accepts status and usage information for MFD/Printers etc. on a corporate network and then periodically and/or automatically places an order for additional supplies, scheduled maintenance etc. (includes mutual authentication and payments/billing)

UC0011 – [DW] Document Creation: A service is available that accepts a request to create a physical embodiment of a document. Special formats (e.g., booklets, bindings), media (e.g., posters, t-shirts) and handling (e.g., FEDEX delivery of results) can be specified. Authentication and payments/billing are supported.

UC0012 – [PP] I have many components that accept some kind of business document. All of these business documents have certain common behaviors and in particular they may be cancelled, approved or postponed. It must be possible for one component (let's say the purchasing interface) to report the existence of a document to another component (let's say the purchasing approval workflow engine). The purchasing approval component must be able to invoke the "cancel", "approve" or "postpone" methods on the business document.

Of course in simple cases one can always work around the lack of references by passing around magic cookies to some kind of "purchase order manager" but this scales very poorly because it means that ALL access to purchase orders throughout the entire company must go through a single component, just because it happens to implement the cancel_based_on_cookie(), approve_based_on_cookie() and postpone_based_on_cookie methods.

In programming terms the right way to do it is:

```
interface po:
  def approve():
    (do some authorization etc. and then change the state to cancel)

  def cancel():
    (do some authorization etc. and then change the state to cancel)

  def postpone():
    (do some authorization etc. and then change the state to cancel)

class purchasing_interface_1:
   def accept_purchase_order(address, items, etc. etc.):
      validate(address, items, etc. etc.)
      po = new PO(address, items, etc. etc.)
      my_list_of_purchase_orders.add(po)
      approval_engine.add(po)
      return po

class purchasing_interface_2:
   # same basic thing but owned by different part of organization
   def our_accept_purchase_order(address, items, etc. etc.):
      validate(address, items, etc. etc.)
      po = new PO(address, items, etc. etc.)
      my_list_of_purchase_orders.add(po)
```

```
        approval_engine.add(po)
        return po


class approval_workflow_component:
    def ceo_says_approve(po):
        po.approve()
```

In programming terms the wrong way to do it is:

```
class po_manager:
   def new_po(address, item, etc.):
       cookie = make_new_cookie()
       my_list_of_purchase_orders.add(cookie, address, item)
       return cookie

   def approve(cookie):
      (do some authorization etc. and then change the state to cancel)

   def cancel(cookie):
      (do some authorization etc. and then change the state to cancel)

   def postpone(cookie):
      (do some authorization etc. and then change the state to cancel)

class purchasing_interface_1:
    def accept_purchase_order(address, items, etc. etc.):
        validate(address, items, etc. etc.)
        po_cookie = global_po_manager.new_po(address,item, etc.)
        approval_engine.add(po)
        return po_cookie

class purchasing_interface_2:
    def accept_purchase_order(address, items, etc. etc.):
        validate(address, items, etc. etc.)
        po_cookie = global_po_manager.new_po(address,item, etc.)
        approval_engine.add(po)
        return po_cookie

class approval_workflow_component:
    def ceo_says_approve(cookie):
        global_po_manager.approve(cookie)
```

Note how instead of just specifying the interface to purchase orders I had to centralize everything through a single component that implemented that interface for purchase orders from different parts of the company that might not otherwise have had to be

centralized. The centralization decision should be a business decision and not forced by the weaknesses of the description language.

WSDL needs to be able to define the abstract "purchase order" interface and to describe methods like accept_purchase_order which return objects of that type. At the SOAP level those would be URIs to dynamically created endpoints representing those purchase orders.

UC0013 – [AK] Inventory reporting. In vendor-managed inventory scenarios, suppliers monitor the inventory levels of their customers then, when levels of items meet specified reorder points, the suppliers restock the inventory to predetermined levels, and bill the customers for the goods actually sold. In the case of some retail food items, delivery drivers make these determinations (next time you are at the supermarket, watch the potato chip delivery drivers go through this process as they restock the racks). But no company wants to carry inventory and all parties in a supply chain would like to be relieved of the guess work and carrying costs that accompany inventories.

With standard UPC/EAN product numbers bar coded on package labels, and inexpensive bar code reading systems, it is possible for even the smallest companies to track inventories closely. However, the companies need the infrastructure to capture the bar code scanning data and report them to suppliers in a timely, consistent, and meaningful way. A Web service that can perform these functions would either regularly (i.e., daily) report inventory levels or alert the suppliers when the customer meets reorder points on specified items.

The description should be able to identify business processes, inventory reporting transaction(s), periodic inventory level report, report frequency such as hourly/daily/weekly, real-time, inventory items such as UPC/EAN product identifiers, inventory item quantities, reorder flag (Yes/No), party identification, e.g. DUNS number, message format specifications, acknowledgement required, authentication process, encryption requirements

UC0014 – [AK] Travel service volume discounts. Cruise lines need to market their services to individuals or couples, which is expensive and risky. Likewise, passengers purchase space on cruise ships individually, which limits their ability to get meaningful discounts. Both customers and cruise lines would benefit from a service that enables individual customers with common preferences to identify those preferences and book their travel as a group. The Open Travel Alliance specifications include a customer profile that can identify and transmit detailed travel preferences.

A Web service that describes passenger cruise preferences, such as dates, destinations, or special features (e.g., Geek Cruises … they really exist),

would enable potential passengers to group together and bid among the cruise lines for their business. The Web service would need to aggregate the individual preferences into groups and describe those preferences for vendors. These processes could be replicated for other auction or reverse-auction transactions.

The web service description should identify business processes, customer inquiry, customer aggregation, travel service offer, transactions: request and response messages for each process like customer inquiry request/response, customer aggregation request/response and travel service offer request/response, party identification, e.g. DUNS number, ATA number, message format specifications, acknowledgement required, authentication process, encryption requirements

UC0015 – [JJM] Request-response: Two parties wish to conduct electronic business by the exchange of business documents. The sending party packages one or more documents into a request message, which is then sent to the receiving party. The receiving party then processes the message contents and responds to the sending party. Examples of the sending party's documents may be purchase order requests, manufacturing information and patient healthcare information. Examples of the receiving party's responses may include order confirmations, change control information and contractual acknowledgements.

UC0016 – [JJM] Remote Procedure Call (RPC): The sender invokes the service by passing parameters that are serialized into a message for transmission to the receiving server.

UC0017 – [JJM] Request with acknowledgement: A sender wishes to reliably exchange data with a receiver. It wishes to be notified of the status of the data delivery to the receiver. The status may take the form of: 1. The data has been successfully delivered to the receiver, or 2. Some failure has occurred which prevents the successful delivery to the receiver.

UC0017 – [JJM] Request with encrypted payload: A sender wishes to exchange data with a receiver and has agreed to encrypt the payload. The sending and receiving applications agree on the encryption methodology. Data is encrypted by the originating application and sent to the receiver via SOAP. The data reaches the receiving application untouched, and may then be decrypted in the agreed-upon manner.

UC0018 – [JJM] Message header and payload encryption: Two trading partners engaged in a message exchange may agree to cryptographically sign and verify either the message header, the routing header(s) and/ or the payload. The sender or originating application may perform the signing of the payload. The sending message handler signs the message header. A routing header may be appended to the message header. The routing header may also be signed by a message service handler.

UC0019 – [JJM] Third party intermediary:A blind auction marketplace serves as a broker between buyers and suppliers. Buyers submit their requirements to the marketplace hub, which broadcasts this information to multiple suppliers. Suppliers respond to the marketplace hub where the information is logged and ultimately delivered to the buyer.

UC0020 – [JJM] Communication via multiple intermediaries: An intermediary forwards a message to the ultimate receiver on behalf of an initial sender. The initial sender wishes to enforce the non-repudiation property of the route. Any intermediate message service handler that appends a routing message must log the routing header information. Signed routing headers and the message readers must be logged at the message handler which passes the message to the ultimate receiver to provide the evidence of non-repudiation.

UC0021 – [JJM] Asynchronous messaging: A sender sends a message asynchronously to a receiver expecting some response at a later time. The sender tags the request with an identifier allowing the response to be correlated with the originating request. The sender may also tag the message with an identifier for another service (other than the originating sender) which will be the recipient of the response.

UC0022 – [JJM] Sending non-XML data: A digital camera wishes to transmit image data over a wireless link using SOAP to a remote server. The binary image data (non-XML) accompanies the message. The digital camera represents a situation in which connections from the receiver to the sender may not be permitted due to device limitations or firewalls.

UC0023 – [JJM] Multiple asynchronous responses: An application requests some information from a server, which is returned at a later time in multiple responses. This can be because the requested information was not available all at once (e.g., distributed web searches).

UC0024 – [JJM] Event notification: An application subscribes to notifications of certain named events from an event source. When such events occur, notifications are sent back to the originating application (first party notification) or to another application (third party notification). For example, an application can subscribe to notification of various aspects of a printer's status (e.g., running out of paper, ink etc.). The notifications of such events could be delivered to a management application

UC0025 – [IS] Service Metadata: A WS provider can decorate various elements of the service description with custom attributes. These attributes may be application specific and would be described by the WS provider in an additional documentation. Such custom attributes may be defined in a specific schema. WS provider may include such extra information as owner e-mail, link to SLA, security and session requirements for a particular message, etc.

Here is an example of extended attribute definitions and inclusion. <descriptions ... > <extend xmlns:myExt="..."> <myExt:owner id="owner1" email="myadmin@mycorp.com/> <myExt:sec id="sec1" signatureRequired="yes"/>

<myExt:sess id="sess1" cookie="MYCTX"/> </extend> <types>... <message extend="sec1 sess1" ... <portType... <binding ... <service extend="owner1" ...

A WS client can interrogate the metadata attributes as follows

NodeList ext = service.getExtend();

Similarly for message descriptions.

UC0026 – [IS] References: A WS provider can define operations that return and/or take as a parameter a reference to another WS interface.

The definition would look as follows

```
<definitions ... xmlns:ref="http://schemas.xmlssoap.org/wsdl/ref">
<message name="...">
        <part name="param" type="ref:ref">
</message>
```

A schema for http://schemas.xmlssoap.org/wsdl/ref is as follows

```
<schema targetNamespace="http://schemas.xmlssoap.org/wsdl/ref"
        xmlns:ref="http://schemas.xmlssoap.org/wsdl/ref">
        <complexType name="ref">
        <all>
                <element name="description" nillable="true" type="xsd:string"/>
                <element name="service" type="xsd:QName"/>
                <element name="port" nillable="true" type="xsd:string"/>
        </all>
        </complexType>
        <element name="ref" type="ref:ref"/>
</schema>
```

Then a WS client can use references to the interfaces as follows

MyExtSvc esvc = new MyExtSvc(service.myMethodReturnungRef(...))

The underlying WS framework would support instantiation of a service based on reference (like most already instantiate based on an endpoint URL).

I believe systinet does something similar, but unless it's mandated by the WSDL standard it is as good as private app-specific extension.

UC0027 – [IS] Sync/Async Operations: To negotiate proper communication sequence WS provider has to be able to describe if certain operations can be handled asynchronously, must be handled asynchronously or synchronously and what is the expected execution time. This would allow process orchestration system to properly adjust the flow and not run into unexpected blocking.

Here is an example of operation definitions.

```
<portType>
        <operation ... handling="sync async" replyTime="10000"/> <!-- up to the client --
>
        <operation ... /> <!-- only sync -->
        <operation ... handling="async" replyTime="unknown"/> <!-- long running,
human dependant --> </portType>
```

WS client would then get to use operations properly. Similar to this.

```
AsyncContext ctx = service.start_myAsyncOp(...);
MyResult result = service.waitFor_myAsyncOp(ctx);
```

The underlying WS framework would then initiate proper SOAP messaging sequence with acknowledgement and notification phases. SOAP protocol must support asynchronous messaging.

UC0028 – [IS] Namespaces with data and interfaces: A service can have an OO model like this
my.app.model.Address is a base class to represent
data my.app.impl.Address inherits my.app.model.Address
my.app.model.AddressBook is an interface my.app.impl.AddressBook is an implementation of my.app.model.AddressBook

It is possible to represent this model in WSDL and associated XML Schema by placing schema and interfaces in the proper XML namespaces. It has to be required that namespaces are not getting lost between service provider and the client. It should be part of WSDL compliance.

Here is a brief example:

```
<definitions
        xmlns:model="urn:my.app.model"
        xmlns:impl="urn:my.app.impl">
<types>
<schema targetNamespace="urn:my.app.model">...
<schema targetNamespace="urn:my.app.impl">...
</types>
<message targetNamespace="urn:my.app.model" ...
<message targetNamespace="urn:my.app.impl" ...
<portType targetNamespace="urn:my.app.model" ...
<portType targetNamespace="urn:my.app.impl" ...
```

UC0029 – [IS] Events: A WS provider can describe events generated by a service as follows

```
<message name="hasDataIn">
	<part name="container" type="data:Container"/>
</message>
<message name="hasDataOut">
	<part name="context" type="data:Context"/>
</message>
<portType>
	<operation...
	<event name="hasData1" mode="poll" interval="10">
		<input message="inerface:hasDataIn"/>
		<output message="inerface:hasDataOut"/>
	</event>
	<event name="hasData2" mode="push">
		<input message="inerface:hasDataIn"/>
		<output message="inerface:hasDataOut"/>
	</event>
</portType>
```

And this way WS client may subscribe to events like this

```
service.subscribe_hasData1(new data.Container(...),new myServiceListener())
service.subscribe_hasData2(new data.Container(...),new myServiceListener())
```

And implement a proper handler

```
class myServiceListener
{
	void hasData1(data.Context ctx) { ... }
	void hasData2(data.Context ctx) { ... }
}
```

The underlying WS framework would take care of the event by either polling (sending a SOAP request) with a specified interval or registering a SOAP listener (endpoint) with the target WS (according to the event definition in WSDL).

We should also describe the SOAP protocol sequence (registrtion/acknowledgement/notification) for the events in accordance with asynchronous SOAP messaging.

UC0030 – [IS] Versioning: A WS provider can describe versions of interfaces implemented by a service. Such as this

```
<definitions xmlns:interface-latest="urn:myService-latest"
	xmlns:interface-ver1="urn:myService-ver1" ... >
<binding targetNamespace="urn:myService-latest" version="2.0.0.0"> ... <binding
targetNamespace="urn:myService-ver1" version="1.0.0.0"> ... <service
```

name="myServiceService"> <port name="myService" binding="interface-latest:myServiceSoapBinding"> ... <port name="myService" binding="interface-ver1:myServiceSoapBinding"> ... </service>

WS client can bind to the necessary interface version. This way there is no ambiguity when WS proivider changes service interfaces and client has created a static proxy that uses previous version of interfaces.

WS provider can deprecate and remove interfaces as desired, and the client would know that. Client would send a SOAP request that would not be accepted (as namespaces do not match), as opposed to client trying to send a SOAP request that could be accepted, but improperly executed.