

Стандартизация определений web-сервисов

Владимир Энгельс,
Oracle СНГ

Источник: сайт SOA Design, статья по SOA, 29.8.2007
<http://soa.it-consultants.ru:8888/?q=node/7>



Предисловие

Повышенная доступность интерфейсов на основе web-сервисов, по сравнению со всеми предыдущими распределенными архитектурами, требует системного подхода к проектированию согласованного, целостного интерфейсного решения. Интерфейс web-сервиса перестает быть внутренним делом системы или решения. Экспорт функций системы на основе жестких внутренних стандартов - залог успешного достижения целей проекта, а так же гарантия возможности поддержки данного решения в будущем.

Данная статья показывает, как дополнительные усилия, потраченные на определение интерфейсов сервисов, помогают в разработке стандартизованных SOA-решений. В ней освещается приблизительный набор соглашений проектного характера для использования внутри WSDL.

Область сервис-ориентированности

Технологические платформы создаваемые в настоящее время на основе модели сервис-ориентированности (SOA) представляют очередной шаг в эволюции распределенных вычислений. То, что лежит в основе SOA (обычно именуемая сервис-ориентированная парадигма), затрагивает как вопросы проектирования отдельных модулей логики автоматизации (сервисы), так и структуру окружения проектируемых сервисов.

Принципы, заложенные в сервис-ориентированность, такие как повторное использование, кроссплатформенность и интегрируемость - влияют на то, каков будет дизайн отдельных сервисов, решений и интеграционных инфраструктур. Этот дизайн, в свою очередь, влияет на взаимодействие отдельных элементов инфраструктуры друг с другом. Хотя SOA-модель позволяет организациям покрывать стандартами интерфейсный уровень сервисов, основная выгода от применения данного подхода наступает только при реализации такого рода стандартов на самых ранних стадиях работы.

Влияние стандартов

Как и в случае с другими платформами или архитектурными моделями, основополагающими принципами можно злоупотребить. И в случае SOA существует очень большая вероятность «застандартизовать» решения ввиду их потенциально очень широкой «области охвата».

В процессе стандартизации этапа проектирования, требуется определить ограничения текущего окружения и противопоставить им цели стратегической SOA-инициативы, такие как охват технических и организационных изменений, приносящие достаточный эффект в развитии. Необходимо учесть так же, что создание стандартов может повлечь расходование значимых ресурсов и времени.

Когда стандартизация завершена, использование стандартов необходимо жестко встроить в жизненные циклы проектов реализации SOA-решений. Принуждение разработчиков использовать стандарты иногда может оказаться более сложным делом, чем создать сам стандарт. Часто наблюдается очень большое сопротивление внедрению стандартов, поскольку они рассматриваются как искусственные правила ограничивающие креативность и «творческий полет» разработчиков, а также делают менее гибким процесс разработки. На практике можно услышать и более экзотические причины, чтобы не использовать стандарты.

В тоже время, упорствующие разработчики могут обнаружить, что стандарты могут серьезно ускорить процесс разработки решения, устанавливая границы, защищающие проектную команду от необходимости постоянно исследовать альтернативные проектные решения. Это может сэкономить массу времени на начальных этапах выполнения проекта, а также упростить дизайн решения в целом.

Естественно, что в процессе реализации инициативы перехода к SOA-решениям уровня предприятия будут накладываться ограничения, как на частные проекты,

так и на отдельных разработчиков. Например, уделение внимания повторному использованию сервисов, может потребовать дополнительных усилий и времени для идентификации тех модулей, которые могут быть успешно использованы за пределами частного проекта.

В тоже время, проекты будут защищены от разработки модулей, функциональность которых может быть реализована, путем повторного использования существующего модуля. В этом случае, разработчик будет вынужден строить свои решения с учетом множества уже реализованных сервисов.

Эти и многие другие обстоятельства могут привести к чувству, что стандарты при реализации SOA-решений являются обузой для частных проектов, так как добавляют сложности в архитектуру решения в целом и негативно сказывается, как на бюджете, так и на сроках исполнения проектов. Основным средством борьбы с этим чувством на сегодня остается пропаганда инициатив по стандартизации в среде разработчиков и менеджеров.

Создание предсказуемого SOA-решения уровня предприятия

Стандарты могут быть применены на различных уровнях SOA-окружения. На самом высоком уровне можно стандартизировать спецификацию архитектуры и другие типы внутренних документов, а также различные компоненты инфраструктуры, такие как реестр сервисов.

В тоже время, стандарты на дизайн, особенно для отдельных сервисов, в большинстве случаев не применяются, хотя и являются критичными с точки зрения успешного построения SOA-решения.

Их значение усиливается в случаях, когда сервис реализуется как web-сервис, поскольку платформы взаимодействия на основе открытых стандартов, которые лежат в основе web-сервисов, существенно расширяют возможности для вызова сервисов, которые могут исходить из любой точки IT-инфраструктуры предприятия. Когда список стандартизованных web-сервисов увеличится, он становится мощным средством повторного использования инвестиций сделанных для уже разработанных и развернутых сервисов.

В настоящий момент, web-сервисы представляют собой передовое средство для достижения стандартизованного SOA-решения. Но, чтобы сделать это, необходимо взять под строгий контроль определение WSDL. Остальная часть статьи расскажет, как определение интерфейсов web-сервисов может быть окружено стандартами для построения SOA-решения.

Контроль над WSDL

Подход «снизу вверх» подразумевает, что разработчик уже

имеет некий интерфейс к модулям бизнес-логики и при формировании определения интерфейсов web-сервисов в основном полагается на утилиты автогенерации, предоставляемые большинством средств разработки.

Подход же «сверху вниз» напротив подразумевает, что разработчик должен меньше полагаться на утилиты автогенерации, а вместо этого должен больше использовать «ручное» определение интерфейсов web-сервисов. Преимущества подхода «сверху вниз» - это высокий уровень контроля, который разработчик имеет над интерфейсом сервисов и инкапсулируемой логикой.

Поскольку в подходе «снизу вверх» определение интерфейсов web-сервисов практически неподконтрольно разработчику, то и возможностей для применения стандартов уровня проектирования интерфейсов web-сервисов здесь нет. Поэтому в дальнейшем повествование в основном будет относиться к подходу «сверху вниз».

Для понимания стандартов уровня определений сервисов, необходимо разбить структуру WSDL, которая используется для определения интерфейса web-сервисов. WSDL – это XML документ, определенный спецификацией W3C WSDL, и содержащий абстрактные и конкретные описатели, которые формально описывают, как интерфейс web-сервиса, так и его реализацию. Эти разделы WSDL разделены, с целью дать возможность вносить изменения в раздел описания реализации, без необходимости внесения изменений в раздел описания интерфейсной части.

Поскольку конкретные определения часто подвержены влиянию или даже определяются существующей технической инфраструктурой, фокус статьи будет направлен на дизайн абстрактного определения, где на передний план выступают следующие соображения проектного характера (используются английские термины с переводом, чтобы сохранить связь с англоязычной литературой):

- *Service interface granularity* (гранулярность интерфейсов сервисов) – логика, инкапсулированная в сервисе, разбивается и экспортируется в форме операций сервиса.
- *Service data exchange requirements* (требования к обмену данными с сервисом) – выставляются специфические требования к сообщениям.
- *Service data representation* (представление данных сервиса) – определяемые структуры данных и типы данных или ссылки на них.
- *Service interface labeling* (именование в интерфейсах сервисов) – соглашения по наименованию, применяемые к элементам сервиса и сервису в целом.

Это основные аспекты проектирования определений сервисов, которые влияют на качество и простоту использования сервисов в целом. Рассмотрим эти аспекты более подробно.



Рисунок 1: Области определения абстрактного WSDL, к которым могут быть применены стандарты.

Гранулярность интерфейсов сервисов

Решение «как инкапсулируемая в сервисе логика должна быть экспортирована через внешний интерфейс» может оказаться самой проблемной частью дизайна сервиса. Это может привести к большим дебатам в среде архитекторов и проектировщиков предприятия на тему «должны ли быть сервисы крупно-гранулярными или мелко-гранулярными».

Мелко-гранулярные операции обычно ассоциируются с шаблоном на основе вызова удаленных процедур (RPC), где фокус делается на обмене параметрами небольшого объема для выполнения специфической задачи. Если необходимо выполнить другую задачу тем же сервисом, вызывается другая операция сервиса. Пример 1 демонстрирует конструкции «сообщение» и «операция», используемые для определения мелкогранулярной операции сервиса.

Пример 1 Операция `GetProfileName` ожидает при запросе единственный входной параметр – «идентификатор профиля», используемый как критерий поиска, при получении результата – «имя профиля». В данном случае входное и результирующее значения представлены примитивными XSD-типами (Для определения мелкогранулярных сообщений могут также использоваться комплексные типы с конструкциями из простых типов.)

```
<definitions ...>
  ...
  <message name="GetProfileNameRequest">
    <part name="ProfileID" type="xsd:int"/>
  </message>
  <message name="GetProfileNameResponse">
    <part name="ProfileName" type="xsd:string"/>
  </message>
  ...
  <operation name="GetProfileName">
    <input message="tns:GetProfileNameRequest"/>
    <output message="tns:GetProfileNameResponse"/>
  </operation>
  ...
</definitions>
```

Хорошо известная обратная сторона мелко-гранулярного подхода выражается в том, что вызов операций сервисов на основе таких сообщений становится очень «дорогостоящим», и что результирующая нагрузка на передачу и обработку отдельных сообщений для выполнения мелких подзадач в сложных процессах становится чрезмерной.

Крупно-гранулярные операции позволяют выполнять больше функций в рамках одного вызова сервиса. Часто такой вызов преобразуется в последовательность вызовов мелких задач внутри крупно-гранулярной операции, позволяя таким образом выполнить больше работы всего за один вызов сервиса, как показано на примере 2.

Пример 2 Операция `GetProfileInfo` использует в запросе множество входных значений (критерии поиска) в форме конструкции комплексного типа. Аналогично, результат выполнения операции в форме конструкции комплексного типа также может содержать множество возвращаемых значений (каждое из которых может быть информацией о профиле). Хотя, данный пример и классифицируется как крупно-гранулярная операция, поскольку она спроектирована с использованием гибкой схемы, данный тип операции может обеспечивать, как крупно-гранулярный обмен данными, так и мелко-гранулярный.

```
<definitions ...>
  ...
  <message name="GetProfileRequest">
    <part name="RequestValue" element="act:
GetProfileRequestType"/>
  </message>
  <message name="GetProfileResponse">
    <part name="ResponseValue" element="act:GetPro-
fileResponse"/>
  </message>
  ...
  <operation name="GetProfileInfo">
    <input message="tns:GetProfileRequest"/>
    <output message="tns:GetProfileResponse"/>
  </operation>
  ...
</definitions>
```

Позитивный аспект крупно-гранулярного подхода – меньшее количество актов взаимодействия клиента и провайдера сервиса, а также меньшее количество циклов обработки. Менее приятное следствие то, что крупно-гранулярная операция часто выполняет больше функций, чем требуется в конкретный момент времени. Как результат, такие операции требуют больше информации или большие структуры документов на входе от всех клиентов, тогда как для выполнения частных запросов используется только малая часть входной информации. Полной же функциональностью крупно-гранулярной операции пользуются в редких случаях или вообще никогда.

Проектирование интерфейсов сервисов достаточно тяжело стандартизировать на одном уровне гранулярности. Поэтому архитекторы или проектировщики должны постоянно

помнить о том, что уровень гранулярности проектируемых сервисов удовлетворяет сбалансированным потребностям, как для того проекта, для которого он проектируется непосредственно, так и для тех проектов, которые появятся в результате обозримого изучения предметной области или гипотетического анализа. Данный баланс гарантирует, что интерфейс сервиса удовлетворяет необходимому уровню гранулярности сервиса, как для текущего, так и для прогнозируемого обмена данными с сервисом.

Требования к обмену данными с сервисом

Когда решен вопрос, какие операции будет содержать сервис, а также какой объем логики будет в нем инкапсулирован, необходимо описать, какие входные и выходные типы данных необходимы и достаточны операциям для выполнения спроектированного объема работы. Сложность данного проектного решения зависит от заложенного уровня гранулярности интерфейса. Мелко-гранулярные операции имеют тенденцию выдвигать фиксированные потребности в обмене простыми типами данных, тогда как крупно-гранулярные – предполагают многообразие комбинаций множеств входных и выходных типов данных.

Общая тактика для избежания проблемы использования крупно-гранулярными операциями редко используемых данных на входе или выходе операции – использование необязательных параметров. Операция может выполнять множество связанных функций, но конкретный набор выполняемых функций определяется конкретным составом входных параметров. Аналогично, конкретные выходные значения, возвращаемые операцией, могут меняться, в зависимости от реально выполненного набора операций.

Хотя данный подход практичен и гибок, все же использование необязательных параметров должно быть ограничено для web-сервисов. Использование необязательных параметров может привести к непрозрачному проекту интерфейса сервиса и к множеству запрещенных комбинаций входных параметров. Вдобавок степень свободы при использовании тех или иных необязательных параметров часто ограничена используемой проектной или «корпоративной» моделью данных (в форме XML-схемы), которые ложатся в основу проектируемых структур. Стандартизованные же модели корпоративных данных, как правило, имеют очень жесткую структуру.

При рассмотрении требований к обмену данными с сервисом, также необходимо принять связанное решение – требуется ли для операций сервиса односторонний или двусторонний обмен данными. В эру RPC архитекторы и разработчики были «пропитаны» шаблоном синхронного обмена данными. Внутри же инфраструктуры, основанной на сообщениях, часто возникает требование выполнения одностороннего обмена данными для поддержания шаблона асинхронного взаимодействия. Аккуратное формулирование стандартов и руководств поможет

проектировщикам и разработчикам сервисов стабильно принимать правильное решение по вопросу синхронности/асинхронности взаимодействия с сервисом.

Представление модели данных сервиса

В сердце любого серьезного определения WSDL находится хорошо определенная XML схема. Для XML схем, которые определяют структуры данных для входных и выходных сообщений сервиса, настолько же важно быть повторно используемой и быть определенной в терминах предметной области или в терминах модели мастер-данных (http://www.oracle.com/global/ru/oramag/dec2007/russia_soa_en1.html) (а не в терминах специфичных бизнес-процессов или конкретных решений), как и для определений WSDL, описывающих логику сервиса. Построение централизованного набора схем модели мастер-данных – типов данных, используемых в различных решениях – позволяет избежать больших накладных расходов, связанных с избыточным проектированием частных моделей данных, а также вносит вклад в создание единой, согласованной и безупречной модели мастер-данных.

Другими словами, правильно стандартизованные интерфейсы сервисов требуют использование в качестве основы правильно стандартизованного определения XML схемы. В то же время, важно отметить, что XML схемы, спроектированные для конкретного определения WSDL, выглядят иначе, чем XML схемы, спроектированные для описания конкретной предметной области и конкретной модели мастер-данных. Например, XML схема, создаваемая специально для web-сервиса, имеет тенденцию включать в себя наименования, которые ссылаются на сообщения запроса и ответа (как показано в примере 3).

Пример 3 Элементы «part» ссылаются на типы данных, которые четко именованы в соответствие с их ролью в данном конкретном web-сервисе.

```
<message name="GetProfileRequest">
  <part name="RequestValue" element="act:GetProfileRequestType"/>
</message>
<message name="GetProfileResponse">
  <part name="ResponseValue" element="act:GetProfileResponseType"/>
</message>
```

С другой стороны, элементы внутри XML схемы, спроектированной для отражения документов из корпоративной предметной области, будут ориентированы только на проблематику предметной области, а не на аспекты взаимодействия с конкретным сервисом (см. пример 4).

Пример 4 Типы данных, на которые ссылаются данные элементы «part», являются более общими. Их назначение становится более понятным, когда они ассоциируются с именем родительского элемента «message».

```
<message name="GetProfileRequest">
  <part name="RequestValue" element="act:Profile"/>
</message>
<message name="GetProfileResponse">
  <part name="ResponseValue" element="act:ProfileRe-
port"/>
</message>
```

Введение стандартов моделирования XML схем модели мастер-данных требует проведения дополнительного аналитического шага, сравнимого с тем, который необходим при проектировании интерфейсов сервисов. Фундаментальное замечание в отношении данного аналитического шага заключается в том, что XML схемы (и модель мастер-данных в целом), представляющие типы данных с потенциалом повторного использования, не должны проектироваться под нужды конкретного web-сервиса, кроме случаев, когда объем повторного использования таких типов данных непосредственно связан с бизнес-логикой, инкапсулируемой в конкретном сервисе.

Именованье при проектировании интерфейсов сервисов

Хорошо определенный сервис инкапсулирует четко выделенную функциональность в рамках ясного контекста. Именованье такие сервисы обычно очень просто, поскольку назначение и область действия сервиса хорошо понимаются проектировщиком. Правильное именованье в интерфейсе сервиса требуется для организации лаконичного (отложенного) взаимодействия проектировщика сервиса с потенциальным потребителем сервиса, что характерно для слабосвязанных решений на основе web-сервисов.

Сервисы, которые не могут похвастаться правильным именованьем внутри интерфейса, могут доставить много проблем. Если функции операций, экспортируемые сервисом, не достаточно точно соотносятся с подразумеваемым в именовании логическим контекстом, необходимо вернуться к поиску наименования, более точно представляющего возможности сервиса (операции).

При проектировании сервисов, может оказаться полезной категоризация сервисов в соответствии с готовыми моделями сервисов. Данные модели уже делают предположение о контексте и границах применимости моделируемых сервисов. Наиболее общие модели сервисов включают:

- сервисы-утилиты;
- сервисы, ориентированные на сущности;
- сервисы, ориентированные на задачи.

Данные три модели сервисов формируют соответственно три контекста, в рамках которых реализуются сервисы.

- Контекст сервисов-утилит встречается при вызове операций сервисов, которые инкапсулируют комплексные

или общесистемные функции, такие как логирование, обработка исключительных ситуаций, уведомление и т. п. Именованье подобных повторно используемых сервисов необходимо производить в соответствии с их специфическим контекстом обработки запросов, избегая терминов, привязанных к каким либо конкретным решениям. Например, сервис-утилиты может быть назван Notify.

- Контекст сервисов, ориентированных на сущность, встречается при вызове операций сервисов, представляющих функции по работе с конкретными бизнес-сущностями (типами модели мастер-данных), такими как Счет или Заказ. Именованье сервисов, ориентированных на сущности, часто предопределяется именем самой сущности. Например, сервис может быть просто назван Invoice или Customer.

- Контекст сервисов, ориентированных на задачу встречается для сервисов, как правило, инкапсулирующих логику исполнения какого либо процесса (и характерен интерфейсов BPEL-процессов). В данном случае, алгоритм, увязывающий вместе группу операций, и есть та активность, которая инкапсулируется внутри сервиса, ориентированного на задачи. В связи с этим, общая практика при именовании таких сервисов – использование глаголов в имени. Например, сервис, ориентированный на задачу, может называться GetProfile или ProfileRetrieval, если данное наименование четко очерчивает область выполняемой задачи.

Как и в случае с именами сервисов, именованье отдельных операций также является предметом стандартизации и написания руководств. При именовании операций также существуют общераспространенные используемые практики.

Например, имя самого сервиса накладывает определенные ожидания на именованье его операций. Поскольку хорошее имя сервиса уже достаточно хорошо устанавливает контекст и область действия операций сервиса, имена операций могут быть упрощенными и не использовать много слов. Возьмем для примера операцию получения предыстории счета для сервиса с именем Invoice. Именованье данную операцию как GetInvoiceHistory даже нет необходимости, поскольку контекст Invoice уже устанавливается именем сервиса. Имя GetHistory будет вполне достаточно.

Другой аспект, который необходимо учитывать при именовании операций, заключается в учете возможностей повторного использования логики, инкапсулированной в той или иной операции и в сервисе в целом. Если потенциал для повторного использования присутствует, то необходимо избегать таких имен для операций, которые учитывают только частный аспект выполняемых функций. Другими словами, часто разумнее заранее выбрать более общее имя для операции с учетом возможного использования операции в будущем. Например, можно вполне уменьшить имя операции, названной как GetCurrentWidgetPrice, до GetCurrentPrice или даже до GetPrice.

В случае организации взаимодействия с сервисом в рамках событийно-ориентированной архитектуры (Event Driven Architecture, EDA) с использованием каналов (“очередь”

или “прямой канал связи”, “темы” или “подписка”, каналы типа “входной порт”, “почтовый ящик”), появляется необходимость в именовании еще двух объектов: каналов и сообщений.

Поскольку в рамках EDA канал является посредником между провайдером события и его потребителем, есть два подхода к именованию каналов: либо в терминах провайдера, либо в терминах потребителя. Подход к именованию каналов в терминах провайдера рекомендуется для каналов типа очередь (один к одному) и тема (один ко многим). Подход к именованию каналов в терминах потребителя рекомендуется для каналов типа “входной порт” (многие к одному) и редко к очередям (один к одному), если необходимо искусственно подчеркнуть семантическую связь очереди с потребителем, например, в случае использования логического именовании входных очередей для процессов. Обособленно стоят каналы типа “почтовый ящик”, именование которых необходимо вести не в терминах провайдера или потребителя, а в терминах отражающих семантику использования столь редкого типа канала связи. Один из примеров именовании канала типа “почтовый ящик” - именование в терминах тех сообщений, накопление которых ведется в канале данного типа. Например, для кластеризации в продукте Oracle Enterprise Service Bus используются каналы типа “почтовый ящик” ESB_ERROR, ESB_MONITOR и др. Запись в такие каналы могут вести все узлы кластера, а считывание - как другие узлы кластера, так и вспомогательные компоненты типа консоли управления, или даже прикладное приложение. Здесь именование каналов произведено в терминах типа событий (ERROR) и в терминах типа процесса, который обеспечивается каналом (MONITOR).

В отношении именовании сообщений, как правило не возникает проблем. Так как сообщение отражает факт наступления какого-либо события, то для именовании событий и соответствующих сообщений рекомендуется использовать глаголы в прошедшем времени с необходимыми пояснениями, например, CreditCard-Verified. Это же правило можно отнести и к цепочкам (последовательностям) событий, наступление которых так же рассматривается как событие (комплексное событие), а также к иерархиям событий.

Соглашения по наименованиям выглядят тривиальными только на первый взгляд, но по мере увеличения репозитория сервисов, появится потенциал для повторного использования и новых вариантов применения сервисов. В больших организациях (или больших и долгоживущих проектах) подразумевается, что со временем очень большое количество архитекторов, аналитиков и разработчиков будут изучать репозиторий сервисов с целью включения их в свои решения. Таким образом, усилия, необходимые для формирования необходимого уровня ясности в отношении всех интерфейсов сервисов, очень быстро окупятся, если интероперабельность и возможность повторного использования тех или иных сервисов будут

легко распознаваться и пониматься всеми участниками.

Заключение

Стандартизация интерфейсов SOA-решения является продолжительным процессом, требующим значительных усилий, а так же большого терпения и дисциплины. Для успешного управления переходом от существующих решений к SOA-решениям, необходимо, чтобы внутри организации (или проекта) сложилось четкое понимание того, как начальные затраты на реализацию стандартов внутри SOA-решения влияют на контроль изменений ИТ-инфраструктуры предприятия.

SOA-решение – начинается с определения сервисов. Уверенность в том, что стандарты применяются от уровня определения интерфейсов сервисов, дает основу для построения прозрачного и управляемого SOA-решения со стабильными интерфейсами, которые в свою очередь стимулируют использование стандартизированной модели данных. Использование стандартизации на ранних этапах проектирования в конечном итоге позволяет успешно создавать очень сложные системы.