# Consumer-Driven Contracts как способ развития сервиса

<https://martinfowler.com/articles/consumerDrivenContracts.html>

* [Проектирование и рефакторинг](https://habr.com/hub/refactoring/),
* [Анализ и проектирование систем](https://habr.com/hub/analysis_design/),
* [Блог компании Сбербанк](https://habr.com/company/sberbank/)
* Черновик
* Перевод



*— Секрет успеха поставщика заключается в том, чтобы дать потребителям качественный товар… ой, то есть сервис. Ну и еще важно не пускаться во все тяжкие с нарушением обратной совместимости.  
Уолтер Уайт*

## От переводчика

### Что это

Это перевод статьи, описывающей шаблон Consumer-Driven Contracts (CDC).  
[Оригинал](https://martinfowler.com/articles/consumerDrivenContracts.html) опубликован на сайте Мартина Фаулера за авторством [Яна Робинсона](http://iansrobinson.com/).

### Зачем это

В микросервисной архитектуре зависимости между сервисами являются источником проблем. Шаблон CDC помогает решать эти проблемы способом, устраивающим и разработчиков сервиса, и его потребителей. Фаулер ссылается на Consumer-Driven Contracts в ключевой статье по микросервисной архитектуре: [Microservices](https://martinfowler.com/articles/microservices.html).

### Для кого это

Статья будет особенно полезна командам, разрабатывающим сервисы для нескольких потребителей в рамках одной организации, и командам-потребителям таких сервисов.

### О терминах

* Я не стал переводить Consumer-Driven Contracts на русский — мы же не переводим в разговоре Test-Driven Development. Если необходимо, можно использовать вариант Контракты, ориентированные на потребителя.
* Provider Contract и Consumer Contract переведены как контракт поставщика и контракт потребителя — они имеют устойчивое использование в русском языке.
* Развитие или эволюция сервиса (service evolution) — доработки, расширение функциональности сервиса.
* Сообщество сервиса (service community) — поставщик плюс все потребители данного сервиса.

## Преамбула

В данной статье обсуждаются проблемы, возникающие между разработчиками и потребителями сервисов, например, когда поставщики меняют часть своего контракта, в частности, схемы документов. Описываются две стратегии борьбы с ними:

1. Добавление точек расширения.
2. "Достаточная" валидация получаемых сообщений.

Обе стратегии помогают защитить потребителей от изменений в контракте, но не дают поставщику сервиса понимания:

* Как их лучше использовать.
* Что необходимо делать по мере развития сервиса.

В статье описан шаблон Consumer-Driven Contracts,  который позволяет поставщикам лучше понимать своих потребителей и фокусирует развитие сервиса на том, что нужно потребителям.

## Пример эволюции сервиса

Чтобы проиллюстрировать некоторые проблемы, с которыми мы сталкиваемся при разработке сервисов, рассмотрим простой сервис ProductSearch, который позволяет осуществлять поиск в нашем каталоге продуктов. Результат поиска имеет следующую структуру:

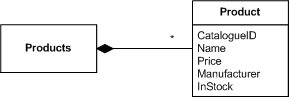


Рисунок 1: Схема результата поиска  
Примерный документ с результатами поиска выглядит следующим образом:

<?xml version="1.0" encoding="utf-8"?>

<Products xmlns="urn:example.com:productsearch:products">

<Product>

<CatalogueID>101</CatalogueID>

<Name>Widget</Name>

<Price>10.99</Price>

<Manufacturer>Company A</Manufacturer>

<InStock>Yes</InStock>

</Product>

<Product>

<CatalogueID>300</CatalogueID>

<Name>Fooble</Name>

<Price>2.00</Price>

<Manufacturer>Company B</Manufacturer>

<InStock>No</InStock>

</Product>

</Products>

Сервис ProductSearch в настоящее время используется двумя приложениями: внутренним маркетинговым приложением и веб-приложением внешнего реселлера. Оба клиента используют XSD для проверки принятых документов до их обработки. Внутреннее приложение использует поля CatalogueID, Name, Price и Manufacturer; внешнее приложение — поля CatalogueID, Name и Price. Ни одно из них не использует поле InStock: хотя оно рассматривалось для маркетингового приложения, оно было выкинуто на раннем этапе разработки.

Одним из наиболее распространенных способов развития сервиса является добавление поля в документ для одного или нескольких потребителей. В зависимости от того, как были реализованы клиентская и серверная части, даже простые изменения, подобные этому, могут иметь дорогостоящие последствия для бизнеса и его партнеров.

В нашем примере, после того, как сервис ProductSearch эксплуатируется в течение некоторого времени, его хочет использовать второй реселлер, но просит добавить поле *Description* к каждому продукту. Из-за того, как устроены клиенты, это изменение имеет значительные и дорогостоящие последствия как для поставщика, так и для существующих потребителей. Стоимость каждого из них зависит от того, как мы реализуем изменение. Есть как минимум два способа, которыми мы можем распределять стоимость изменений между участниками "сообщества сервиса". Во-первых, мы могли бы изменить нашу исходную схему и потребовать, чтобы каждый потребитель обновил свою копию схемы, чтобы правильно проверить результаты поиска; стоимость изменения системы здесь распределяется между поставщиком — который, столкнувшись с таким запросом на изменение, всё равно будет делать какие-то изменения — и потребителями, которых не интересует обновленная функциональность. С другой стороны, мы могли бы добавить вторую операцию и схему для нового потребителя и сохранить исходные операцию и схему для существующих потребителей. Все доработки теперь на стороне поставщика, но сложность сервиса и стоимость его поддержки увеличиваются.

## Интерлюдия: утомленные SOA

Главными преимуществами использования сервисов являются:

1. Повышение организационной гибкости.
2. Cнижение общих затрат на реализацию изменений.

SOA повышает гибкость за счет размещения бизнес-функций в выделенных, переиспользуемых сервисах. Затем эти сервисы используются и оркестрируются для выполнения основных бизнес-процессов. Это снижает стоимость изменений за счет сокращения зависимостей между сервисами, позволяя им быстро перестраиваться и настраиваться в ответ на изменения или незапланированные события.

Однако бизнес может полностью реализовать эти преимущества, только если имплементация SOA позволяет сервисам эволюционировать независимо друг от друга. Чтобы повысить независимость, мы создаем контракты сервисов. Несмотря на это, мы вынуждены дорабатывать потребителей так же часто, как и поставщика, главным образом потому, что потребители зависят от конкретной версии контракта поставщика. В конце концов, поставщики начинают крайне осторожно подходить к изменению любого элемента контракта; это, в частности, потому, что они не имеют представления о том, как потребители реализуют этот контракт. В худшем случае, потребители завязываются на поставщика, описывая всю схему документа в рамках своей внутренней логики.

Контракты обеспечивают независимость сервисов; парадоксально, что они могут также связывать поставщиков и потребителей нежелательными способами. Не задумываясь о функции и роли контрактов, которые мы реализуем в нашей SOA, мы подвергаем наши сервисы «скрытой» связи. Отсутствие какой-либо информации о том, как потребители сервиса реализуют контракт в коде, а также отсутствие ограничений на реализацию (и для поставщика, и для потребителя), в совокупности подрывают предполагаемые преимущества SOA. Короче говоря, предприятие начинает утомляться от сервисов.

## Версионирование схемы

Мы можем начать исследование проблем контракта и зависимостей, мешающих нашему сервису ProductSearch, с версионирования схемы. [Группа технической архитектуры WC3 (TAG)](https://www.w3.org/2001/tag/doc/versioning) описала ряд стратегий управления версиями, которые могут помочь нам разработать схемы для наших сервисов способами, которые уменьшают проблемы с зависимостью. Эти стратегии варьируются от чрезмерно либеральной none, которая требует, чтобы сервисы не различали разные версии схемы и принимали все изменения, до чрезвычайно консервативного большого взрыва, что требует от сервиса выдавать ошибку, если он получает неожиданную версию сообщения.

Обе крайности приносят с собой проблемы, которые не дают получить ценность для бизнеса и увеличивают общую стоимость владения системой. Явные и неявные стратегии «без версий» приводят к системам, которые непредсказуемы в своих взаимодействиях, плохо развиваемы и имеют высокую стоимость доработок. С другой стороны, стратегии "большого взрыва" создают сильно связанные сервисные ландшафты, в которых изменения схемы затрагивают всех поставщиков и потребителей, ухудшая доступность, замедляя развитие и уменьшая возможности для получения прибыли.

"Сообщество сервиса" из нашего примера эффективно реализует стратегию большого взрыва. Учитывая затраты, связанные с повышением ценности системы для бизнеса, ясно, что поставщики и потребители выиграют от более гибкой стратегии управления версиями — той, что TAG называет стратегией совместимости. Она обеспечивает прямую и обратную совместимость схем. При развитии сервисов схемы с обратной совместимостью позволяют потребителям новых схем принимать экземпляры старой схемы: поставщик, созданный для обработки обратно совместимых новых версий, может, тем не менее, принять запрос в формате старой схемы. С другой стороны, прямая совместимость позволяет потребителям старых схем обрабатывать экземпляр новой схемы. Это ключевой момент для существующих пользователей ProductSearch: если бы схема результатов поиска изначально была сделана с учетом прямой совместимости, потребители могли бы обрабатывать ответы новой версии, не нуждаясь в доработке.

### Точки расширения

Составление схем c обратной и прямой совместимостью — это хорошо понятная задача дизайна, лучше всего выраженная шаблоном расширяемости Must Ignore (см. статьи [Дэвида Орчарда](https://www.w3.org/2001/tag/doc/versioning-xml) и [Дейра Обасанжо](https://www.xml.com/pub/a/2004/10/27/extend.html)). Шаблон Must Ignore рекомендует, чтобы схемы включали точки расширения, которые позволяют добавлять элементы к типам и дополнительные атрибуты для каждого элемента. В шаблоне также рекомендуется, чтобы XML описывал модель обработки, которая определяет, что потребителям делать с расширениями. Простейшая модель требует от потребителей игнорировать элементы, которые они не распознают — отсюда и название шаблона. Модель также может потребовать от потребителей обработки элементов, имеющих флаг «Must Understanding», либо выдать ошибку, если они не могут их понять.

Вот наша изначальная схема документа с результатами поиска:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns="urn:example.com:productsearch:products"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

targetNamespace="urn:example.com:productsearch:products"

id="Products">

<xs:element name="Products" type="Products" />

<xs:complexType name="Products">

<xs:sequence>

<xs:element minOccurs="0" maxOccurs="unbounded" name="Product" type="Product" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Product">

<xs:sequence>

<xs:element name="CatalogueID" type="xs:int" />

<xs:element name="Name" type="xs:string" />

<xs:element name="Price" type="xs:double" />

<xs:element name="Manufacturer" type="xs:string" />

<xs:element name="InStock" type="xs:string" />

</xs:sequence>

</xs:complexType>

</xs:schema>

Давайте теперь откатимся назад во времени, и с самого начала укажем для нашего сервиса совместимую, расширяемую схему:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns="urn:example.com:productsearch:products"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

targetNamespace="urn:example.com:productsearch:products"

id="Products">

<xs:element name="Products" type="Products" />

<xs:complexType name="Products">

<xs:sequence>

<xs:element minOccurs="0" maxOccurs="unbounded" name="Product" type="Product" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Product">

<xs:sequence>

<xs:element name="CatalogueID" type="xs:int" />

<xs:element name="Name" type="xs:string" />

<xs:element name="Price" type="xs:double" />

<xs:element name="Manufacturer" type="xs:string" />

<xs:element name="InStock" type="xs:string" />

<xs:element minOccurs="0" maxOccurs="1" name="Extension" type="Extension" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Extension">

<xs:sequence>

<xs:any minOccurs="1" maxOccurs="unbounded" namespace="##targetNamespace" processContents="lax" />

</xs:sequence>

</xs:complexType>

</xs:schema>

Эта схема включает дополнительный элемент расширения для каждого продукта. Сам элемент расширения может содержать один или несколько элементов из целевого пространства имен:

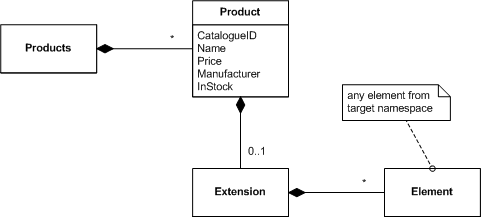


Рисунок 2: Расширяемая схема результата поиска

Теперь, когда мы получаем запрос на добавление описания продукта, мы можем опубликовать новую схему с дополнительным элементом Description, который поставщик вставляет в точку расширения. Это позволяет сервису ProductSearch возвращать результаты, содержащие описания продуктов, и потребители используют новую схему, для проверки всего документа. Потребители, использующие старую схему, не сломаются, хотя они не будут обрабатывать поле "Описание". Новый документ с результатом поиска выглядит следующим образом:

<?xml version="1.0" encoding="utf-8"?>

<Products xmlns="urn:example.com:productsearch:products">

<Product>

<CatalogueID>101</CatalogueID>

<Name>Widget</Name>

<Price>10.99</Price>

<Manufacturer>Company A</Manufacturer>

<InStock>Yes</InStock>

<Extension>

<Description>Our top of the range widget</Description>

</Extension>

</Product>

<Product>

<CatalogueID>300</CatalogueID>

<Name>Fooble</Name>

<Price>2.00</Price>

<Manufacturer>Company B</Manufacturer>

<InStock>No</InStock>

<Extension>

<Description>Our bargain fooble</Description>

</Extension>

</Product>

</Products>

Пересмотренная схема выглядит так:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns="urn:example.com:productsearch:products"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

targetNamespace="urn:example.com:productsearch:products"

id="Products">

<xs:element name="Products" type="Products" />

<xs:complexType name="Products">

<xs:sequence>

<xs:element minOccurs="0" maxOccurs="unbounded" name="Product" type="Product" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Product">

<xs:sequence>

<xs:element name="CatalogueID" type="xs:int" />

<xs:element name="Name" type="xs:string" />

<xs:element name="Price" type="xs:double" />

<xs:element name="Manufacturer" type="xs:string" />

<xs:element name="InStock" type="xs:string" />

<xs:element minOccurs="0" maxOccurs="1" name="Extension" type="Extension" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Extension">

<xs:sequence>

<xs:any minOccurs="1" maxOccurs="unbounded" namespace="##targetNamespace" processContents="lax" />

</xs:sequence>

</xs:complexType>

<xs:element name="Description" type="xs:string" />

</xs:schema>

Обратите внимание, что первая версия расширяемой схемы совместима со второй, а вторая — обратно совместима с первой. Однако эта гибкость происходит за счет повышения сложности. Расширяемые схемы позволяют нам делать непредвиденные изменения, но они дают возможности, которые могут быть никогда не востребованы; при этом мы теряем:

* выразительность, исходящую от простого дизайна
* понятное представление данных, вводя метаинформационные элементы контейнера.

Далее мы не будем обсуждать расширяемые схемы. Достаточно сказать, что точки расширения позволяют нам делать прямо и обратно совместимые изменения в схемах и документах, не затрагивая поставщиков и потребителей сервиса. Однако расширение схемы не помогает нам управлять эволюцией системы, когда нам нужно сделать изменение, нарушанющие контракт.

## Нарушая совместимость



— Да, наша команда нарушила обратную совместимость в последнем релизе! Просто не смогли удержаться от небольшой оптимизации протокола… Ну не обижайся, пупсик!Карла Борин

Наш сервис ProductSearch включает в результаты поиска поле, указывающее, есть ли в наличии данный товар. Сервис заполняет это поле, используя дорогостоящий вызов в древнюю систему инвентаризации — зависимость, которую дорого поддерживать. Поставщик хотел бы удалить эту зависимость, очистить дизайн и улучшить общую производительность системы — желательно, без влияния на потребителей. Общаясь с потребителями, команда поставщика выясняет, что ни одно из потребительских приложений на самом деле ничего не делает с этим полем; хотя и дорогостоящее, оно избыточно.

К сожалению, в текущем положении, если мы удалим поле InStock из нашей расширяемой схемы, мы сломаем существующих потребителей. Чтобы исправить поставщика, мы должны исправить всю систему: когда мы удаляем функциональность от поставщика и публикуем новый контракт, каждое приложение-потребитель нужно будет переустановить с новой схемой, а взаимодействие между сервисами тщательно протестировать. Сервис ProductSearch в этом отношении не может развиваться независимо от потребителей: поставщик и потребители должны все "прыгать одновременно".

Наше сообщество сервиса разочаровывается в его эволюции, потому что каждый потребитель имеет «скрытую» зависимость, которая отражает весь контракт поставщика во внутренней логике потребителя. Потребители, используя валидацию по XSD и, в меньшей степени, статическую привязку к схеме документа в коде, неявно принимают весь контракт поставщика, независимо от их намерений использовать лишь часть.

Дэвид Орчард дает некоторые подсказки относительно того, как мы могли бы избежать этой проблемы, когда ссылается на [принцип надежности](https://en.wikipedia.org/wiki/Robustness_principle): «Реализация должна быть консервативной в своем поведении и либеральной в том, что принимает от других». Мы можем усилить этот принцип в контексте развития сервиса, заявив, что получатели сообщений должны выполнять «достаточную» проверку: то есть они должны обрабатывать только используемые ими данные, и должны выполнять явно ограниченную или целенаправленную проверку полученных ими данных — в отличие от неявно неограниченной валидации «все или ничего», присущей обработке XSD.

### Schematron

Один из способов, которым мы можем настроить проверку на стороне потребителя — это использование масок или регулярных выражений для путей до элементов документа, возможно, с использованием языка валидации структуры дерева, такого как [Schematron](http://www.schematron.com/). Используя Schematron, каждый пользователь сервиса ProductSearch может задать, что он ожидает найти в результатах поиска:

<?xml version="1.0" encoding="utf-8" ?>

<schema xmlns="http://www.ascc.net/xml/schematron">

<title>ProductSearch</title>

<ns uri="urn:example.com:productsearch:products" prefix="p"/>

<pattern name="Validate search results">

<rule context="\*//p:Product">

<assert test="p:CatalogueID">Must contain CatalogueID node</assert>

<assert test="p:Name">Must contain Name node</assert>

<assert test="p:Price">Must contain Price node</assert>

</rule>

</pattern>

Реализации Schematron обычно преобразуют схему Schematron, такую как эта, в XSLT-преобразование, которое получатель сообщения может применить к документу для проверки на корректность.

Обратите внимание, что эта схема не содержит предположений об элементах в исходном документе, которые не нужны приложению-потребителю. Таким образом, явно описана валидация только требуемых элементов. Изменения схемы исходного документа не будут отвергнуты при валидации, если они не нарушают явные ожидания, описанные в схеме Schematron, даже если это удаление ранее обязательных элементов.

Вот относительно легкое решение наших проблем с контрактом и зависимостью, и это не требует от нас добавления неявных метаинформационных элементов в документ. Итак, давайте снова откатимся назад во времени и восстановим простую схему, описанную в начале статьи. Но на этот раз мы также будем настаивать на том, что потребители свободны в своем поведении и валидируют и обрабатывают только ту информацию, которая им необходима (используя схемы Schematron, а не XSD для проверки принятых сообщений). Теперь, когда поставщик добавляет описание для продукта, он может опубликовать пересмотренную схему, не затрагивая существующих потребителей. Аналогичным образом, обнаружив, что поле InStock не проверяется или не обрабатывается ни одним из потребителей, сервис может пересмотреть схему результатов поиска — снова, не затрагивая потребителей.

В конце этого процесса схема ответа ProductSearch выглядит так:

<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns="urn:example.com:productsearch:products"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

targetNamespace="urn:example.com:productsearch:products"

id="Products">

<xs:element name="Products" type="Products" />

<xs:complexType name="Products">

<xs:sequence>

<xs:element minOccurs="0" maxOccurs="unbounded" name="Product" type="Product" />

</xs:sequence>

</xs:complexType>

<xs:complexType name="Product">

<xs:sequence>

<xs:element name="CatalogueID" type="xs:int" />

<xs:element name="Name" type="xs:string" />

<xs:element name="Price" type="xs:double" />

<xs:element name="Manufacturer" type="xs:string" />

<xs:element name="Description" type="xs:string" />

</xs:sequence>

</xs:complexType>

</xs:schema>

## Consumer-Driven Contracts

Использование Schematron в приведенном выше примере приводит к некоторым интересным выводам о контрактах между поставщиками и потребителями, выходящим за рамки проверки документов. В этом разделе мы выделяем и обобщаем некоторые из этих идей и выражаем их в терминах шаблона, который мы называем Контракты потребителей.

Первое, что нужно отметить, это то, что схемы документов являются лишь частью того, что поставщик сервиса должен предлагать потребителям, чтобы они могли использовать его функциональность. Мы называем эту предназначенную для внешних потребителей информацию контрактом поставщика.

### Контракты поставщиков

Контракт поставщика описывает функциональность сервиса в виде набора экспортируемых элементов, необходимых для использования этой функциональности. С точки зрения развития сервиса, контракт представляет собой контейнер для набора экспортируемых элементов, описывающих бизнес-функции. Список этих элементов включает:

* **Схемы документов**. Мы уже подробно обсудили схемы документов. Наряду с интерфейсами, схемы документов являются частями контракта поставщика, изменение которых наиболее вероятно по мере развития сервиса; но, возможно, из-за этого, они также являются частями, с которыми мы имеем больше всего опыта внедрения с различными стратегиями развития сервисов, такими, как точки расширения и использование масок для путей к элементам документа.
* **Интерфейсы**. В простейшей форме интерфейсы поставщика включают набор экспортируемых сигнатур операций, которые потребитель может использовать для управления поведением поставщика. Системы обмена сообщениями обычно экспортируют относительно простые сигнатуры операций и помещают бизнес-содержимое внутрь сообщений, которые они обменивают. В таких системах принятые сообщения обрабатываются в соответствии с семантикой, закодированной в заголовке сообщения или в полезной нагрузке. RPC-подобные сервисы, с другой стороны, кодируют большую часть своей бизнес-семантики в сигнатурах своих операций. В любом случае, потребители зависят от некоторой части интерфейса поставщика, и, следовательно, мы должны учитывать интерфейс при развитии нашего сервиса.
* **Диалоги**. Поставщики и потребители обмениваются сообщениями в последовательностях, которые составляют один или несколько шаблонов обмена сообщениями, таких как "запрос-ответ" или "fire-and-forget". В ходе диалога потребитель может ожидать, что поставщик будет находиться в некотором конкретном контексте при отправке и получении сообщений. Например, сервис бронирования отелей может предложить потребителям возможность зарезервировать номер в начале диалога, а также подтвердить бронирование и внести депозит в последующих обменах сообщениями. Потребитель здесь может разумно ожидать, что сервис «запомнит» детали резервирования при проведении этих последующих обменов, вместо того, чтобы требовать, чтобы стороны повторяли весь "разговор" на каждом этапе процесса. По мере развития сервиса, набор "диалоговых гамбитов", доступных поставщику и потребителю, может измениться. Таким образом, диалоги являются кандидатами на то, чтобы считаться частью контракта поставщика.
* **Политики**. Помимо экспорта схем документов, интерфейсов и диалогов, поставщики могут объявлять и применять конкретные требования к использованию, которые определяют, как можно реализовать другие элементы контракта. Чаще всего эти требования относятся к контекстам безопасности и транзакций, в которых потребитель может использовать функциональность сервиса. Web-сервисы могут выражать это с использованием общей модели WS-Policy и дополнительных спецификаций, таких как WS-SecurityPolicy, но в контексте рассматриваемых нами "политик как кандидатов для включения в контракт поставщика", наше определение политики не привязано к конкретной спецификации или реализации.
* **Характеристики качества обслуживания**. Польза, которую приносят бизнесу поставщики и потребители сервиса, часто оценивается в контексте специфических характеристик качества обслуживания, таких как доступность, латентность и пропускная способность. Мы должны учитывать эти характеристики как вероятные составляющие контракта поставщика и учитывать их в наших стратегиях развития сервиса.

Это определение контракта немного шире, чем то, которое мы обычно подразумеваем, говоря о сервисах, но, с точки зрения развития сервиса оно полезно. Мы не должны считать его исчерпывающим в отношении типов элементов, которые может содержать контракт поставщика: это просто логичный набор элементов, которые являются кандидатами на включение в стратегию развития сервиса. С логической точки зрения, этот набор элементов-кандидатов открыт, но на практике внутренние или внешние факторы, такие как требования к совместимости или ограничения платформы, могут ограничивать тип элементов, которые может содержать контракт. Например, контракт сервиса, который соответствует профилю WS-Basic, не будет содержать политик.

Невзирая на любые подобные ограничения, область контракта определяется просто связностью его элементов. Контракт может содержать много элементов и быть широким по охвату, или сосредоточиться лишь на немногих, до тех пор, пока этого достаточно для описания его функциональности.

Как мы решаем, включать ли элемент в контракт поставщика? Мы делаем это, спрашивая себя, может ли любой из наших потребителей сформулировать (например, в виде тестов) одно или несколько ожиданий в отношении функциональности, описываемой элементом. Мы уже видели из нашего примера сервиса, как потребители могут проявлять интерес к частям схемы документа, экспортируемой сервисом, и как они могут с помощью тестов утверждать, что их ожидания относительно этого контрактного элемента по-прежнему выполняются. Поэтому схема документа является частью нашего контракта поставщика.

Контракты поставщиков имеют следующие характеристики:

* **Закрытые и полные**. Контракты поставщика выражают функциональность сервиса через полный набор доступных для потребителей экспортируемых элементов, и, как таковые, являются закрытыми и полными в отношении доступных для использования функциональных возможностей.
* **Единственные и обязательные**. Контракты поставщика являются единственным и обязывающим описанием функциональности сервиса.
* **Ограниченная стабильность и неизменность**. Контракт поставщика стабилен и неизменен в течение ограниченного периода и/или области действия (см. раздел "Validity of Data in Bounded Space and Time" в статье Пэта Хелланда [Data on the Outside vs. Data on the Inside](http://cidrdb.org/cidr2005/papers/P12.pdf)). Контракты поставщиков обычно используют некоторую форму версионирования, чтобы различать потребителей разных версий контракта.

### Контракты потребителей

Раз мы решили учитывать ожидания потребителей относительно схем, которые мы раскрываем при разработке нашего сервиса — и считаем, что наш поставщик знает о них — тогда нам нужно импортировать эти потребительские ожидания в поставщика. Утверждения Schematron в нашем примере очень похожи на такие тесты. Будучи использованы поставщиком, тесты могут помочь гарантировать, что поставщик продолжает выполнять свои обязательства перед потребителями. Внедряя эти тесты, поставщик получает лучшее представление о том, как он может развивать формат сообщений, которые он создает, не "ломая" функциональность, используемую сообществом сервиса. И там, где предлагаемое изменение затрагивает одного или нескольких потребителей, поставщик будет иметь непосредственное представление об этой проблеме и ему будет проще обсуждать её со всеми заинтересованными сторонами, в зависимости от бизнес-факторов либо учитывая их требования, либо стимулируя их доработки.

В нашем примере мы можем сказать, что набор утверждений/тестов, порожденных всеми потребителями, выражает обязательную структуру сообщений в течение некоторого периода (то есть пока тесты актуальны). Если поставщик обладал этим набором утверждений, он мог бы гарантировать, что каждое отправленное им сообщение валидно для каждого потребителя, поскольку набор утверждений является валидным и полным.

Обобщая эту структуру, мы можем отличить то, что мы уже назвали контрактом поставщика, от индивидуальных контрактных обязательств, которые возникают между поставщиком и потребителем, которые мы теперь будем называть **контрактами потребителей**. Когда поставщик принимает разумные ожидания, выраженные потребителем, он заключает контракт с потребителем.

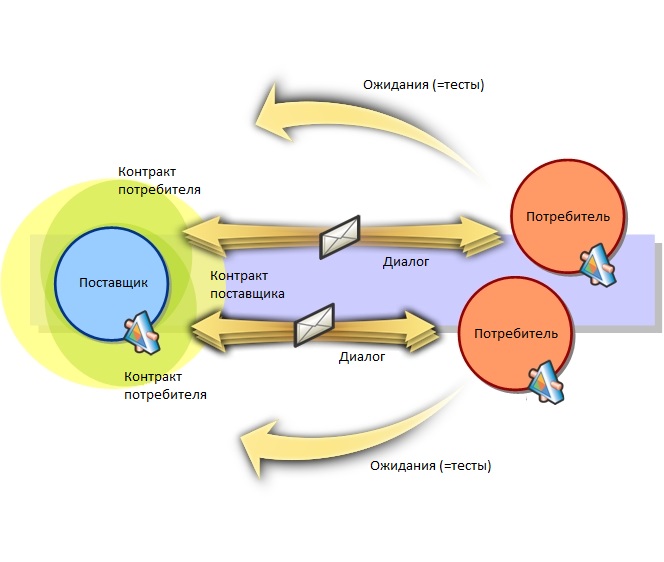


Рисунок 3: Контракты потребителей

Контракты потребителей имеют следующие характеристики:

* **Открытые и неполные**. Контракты потребителей являются открытыми и неполными в отношении функциональности сервиса. Они выражают подмножество возможностей сервиса с точки зрения ожиданий потребителя.
* **Множественные и не обязывающие**. Число контрактов потребителей пропорционально числу потребителей сервиса, и каждый из них не является обязывающим в отношении всего набора обязательств поставщика. Не-обязывающий характер отношений от потребителя к поставщику является одной из ключевых особенностей, которые отличают сервис-ориентированную архитектуру от архитектуры распределенного приложения. Потребители должны понимать, что другие потребители сервиса могут использовать его способами, совершенно отличными от их собственных. Потребители могут развиваться с разной скоростью и требовать изменений поставщика, которые потенциально могут нарушать зависимости и ожидания других потребителей. Потребитель не может предвидеть, как или когда другой потребитель будет нарушать контракт поставщика; у клиента в распределенном приложении нет таких проблем.
* **Ограниченная стабильность и неизменность**. Как и контракты поставщиков, контракты потребителей действительны в течение определенного периода времени и/или области действия.

### Consumer-Driven Contracts

Контракты потребителей позволяют нам думать о ценности для бизнеса в любой момент жизненного цикла поставщика. Выражая ожидания от контракта поставщика, контракты потребителей эффективно определяют, какие части контракта поставщика в настоящее время имеют ценность для бизнеса, а какие нет. Это приводит нас к предположению, что сервисные сообщества могут извлечь выгоду из того, что они сразу будут специфицировать контракт с точки зрения потребителей. С этой точки зрения, контракты поставщиков возникают для удовлетворения ожиданий и требований потребителей. Чтобы отразить производный характер этой новой контрактной договоренности, мы называем такие контракты поставщиков Consumer-Driven Contracts или производными контрактами.

Производный характер этих контрактов-поставщиков-для-потребителей, добавляет гетерогенный аспект к отношениям между поставщиком сервиса и его потребителем. То есть, поставщики подчиняются обязательству, которое возникает за пределами их границ. Это никоим образом не влияет на принципиально автономный характер их реализации; он просто делает явным тот факт, что сервисы успешны, только когда их используют.

Consumer-Driven Contracts имеют следующие характеристики:

* **Закрытые и полные**. Контракт закрыт и завершен в отношении всего набора функций, требуемых от него его существующими потребителями. Контракт представляет собой обязательный набор элементов, необходимых для поддержки ожиданий/тестов потребителей, пока эти ожидания/тесты актуальны для приложений потребителей.
* **Единственные и не обязывающие**. Контракт является единственным, но не обязывающим описанием функциональности сервиса, поскольку он основаны на объединении существующих ожиданий потребителей.
* **Ограниченная стабильность и неизменность**. Контракт является стабильным и неизменным в отношении конкретного набора контрактов потребителей. Иными словами, мы можем провалидировать Consumer-Driven Contract лишь против определенного набора потребительских контрактов, что фактически ограничивает прямую и обратную совместимость контракта во времени и пространстве. Совместимость контракта остается стабильной и неизменной для определенного набора потребительских контрактов и ожиданий, но может быть изменена по мере того, как ожидания приходят и уходят.

### Сводная таблица характеристик контрактов

В следующей таблице приведены характеристики трёх типов контрактов, описанных в этой статье:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Контракт | Открыт | Полон | Количество | Обязателен | Ограничение действия |
| Поставщика | Закрыт | Полный | Один | Обязывающий | Время/область |
| Потребителя | Открыт | Не полный | Много | Не обязывающий | Время/область |
| Для потребителей | Закрыт | Полный | Один | Не обязывающий | Потребители |

### Реализация

Шаблон Consumer-Driven Contracts рекомендует создавать сообщества сервисов, используя потребительские контракты и Consumer-Driven Contracts. Однако в шаблоне не указывается, какие формы или структуры должны принимать эти контракты, а также не определяется, как ожидания/тесты потребителей сообщаются поставщику и проверяются в его жизненном цикле.

Контракты могут быть выражены и структурированы несколькими способами. В своей простейшей форме потребительские ожидания могут быть зафиксированы в электронной таблице или подобном документе и реализованы на этапах проектирования, разработки и тестирования приложения-поставщика. Пойдя немного дальше и сделав unit-тесты, которые подтверждают каждое ожидание, мы можем гарантировать, что контракты описываются и проверяются в автоматическом режиме с каждой сборкой. В более сложных реализациях ожидания могут быть выражены как утверждения Schematron или через WS-Policy, и оцениваются во время выполнения во входных и выходных обработчиках сервиса.

Как и в случае со структурой контрактов, у нас есть несколько вариантов, когда речь идет о способе коммуникации между поставщиками и потребителями. Так как шаблон Consumer-Driven Contracts не привязан к реализации, мы могли бы, если это позволяет организационная структура, передавать ожидания, просто разговаривая с другими командами или используя электронную почту. Если число ожиданий и/или потребителей слишком велико для управления таким образом, мы можем рассмотреть внедрение автоматизации. Каким бы ни был механизм, скорее всего, эти коммуникации будут проводиться до любых разговоров о бизнес-функциях сервиса.

### Преимущества

Consumer-Driven Contracts предлагают две существенные выгоды, когда речь заходит о развитии сервисов. Во-первых, они фокусируют спецификацию и реализацию сервиса вокруг пользы для бизнеса. Сервис имеет ценность для бизнеса только в той степени, в которой он потребляется. Consumer-driven contracts связывают развитие сервиса с пользой для бизнеса, обозначая ценность элементов контракта, экспортируемых потребителями сервисов — того, что потребители требуют от поставщиков, чтобы выполнять свою работу. В результате поставщики выставляют "экономный" контракт, который четко соответствует бизнес-целям, реализуемым потребителями. Изменение — развитие сервиса — возникает только тогда, когда потребители выражают явную потребность.

Разумеется, возможность начинать с минимального набора требований и развивать наш сервис по запросу потребителей, предполагает что мы в состоянии контролируемым и эффективным образом дорабатывать и развертывать сервис и управлять им. Здесь шаблон Consumer-Driven Contractsобеспечивает второе ключевое преимущество. Consumer-driven contracts поставщика дают нам тонкое понимание и быструю обратную связь, необходимые для планирования изменений и оценки их воздействия на приложения, которые уже эксплуатируются. На практике это позволяет нам ориентировать отдельных потребителей и предоставлять им стимулы, чтобы отказаться от ожиданий, которые мешают нам сделать изменения, нарушающие прямую и обратную совиместимость, когда эти изменения назрели. Строя сервис на основе контрактов потребителей, мы наполняем его репозиторием знаний и механизмом обратной связи, на которые мы можем полагаться во время эксплуатации сервиса.

### Риски и ограничения

В этой статье мы определили мотив для введения Consumer-Driven Contracts в сервисный ландшафт, а затем описали, как шаблон Consumer-Driven Contracts учитывает факторы, определяющие развитие сервиса. В завершение обсудим область применимости шаблона, а также некоторые проблемы, которые могут возникнуть при реализации потребительских контрактов и CDC.

Шаблон Consumer-Driven Contracts применим в контексте как одной организации, так и закрытого, тесно связанного сообщества сервисов: конкретнее, в среде, в которой поставщики могут оказывать определенное влияние на то, как потребители заключают с ними контракты. Независимо от того, насколько облегчен механизм передачи и представления ожиданий и обязательств, поставщики и потребители должны знать, принимать и использовать согласованный набор каналов и конвенций. Это неизбежно добавляет уровень сложности и зависимости от протокола к уже и без того сложной сервисной инфраструктуре.

Мы предположили, что системы, основанные на Consumer-Driven Contracts, лучше способны управлять изменениями, нарушающими совместимость. Но мы не собираемся предполагать, что шаблон — это панацея для проблемы с нарушением совместимости: когда все сказано и сделано, несовместимость все еще является несовместимостью. Однако мы считаем, что шаблон дает понимание того, что на самом деле представляет собой это нарушение, и это понимание может служить основой для стратегии управления версионированием сервиса. Более того, как мы уже обсуждали, сервисные сообщества, которые реализуют шаблон, лучше прогнозируют последствия изменений сервиса. В частности, команда поставщика может более эффективно планировать свои стратегии изменений — возможно, помечая элементы контракта как deprecated в течение определенного периода, одновременно стимулируя потребителей к переходу на новые версии контракта.

Consumer-Driven Contracts не обязательно уменьшают зависимости между сервисами. Слабо-связанные сервисы относительно независимы друг от друга, но, тем не менее, остаются связанными. Однако, что шаблон делает, так это выставляет на свет некоторые из этих остаточных «скрытых» зависимостей, чтобы поставщики и потребители могли лучше вести переговоры и управлять такими зависимостями.

Мы обсудили, как потребительские контракты и Consumer-Driven Contracts выражают ценности бизнеса. Но мы должны четко указать, что мы не рассматриваем такие контракты как индекс или показатель бизнес-ценности — они не являются бизнес-метрикой — и, несмотря, на некоторые поверхностные сходства с такими спецификациями, как WS-Agreement и WSLA, они не предназначены для выражения SLA. Основное допущение здесь заключается в том, что сервисы сами по себе не представляют никакой ценности для бизнеса; ценно их использование. Определяя сервисы как можно ближе к месту использования — силами потребителей — мы стремимся создавать пользу для бизнеса в минималистичной, "точно в срок" манере.

Наконец, мы должны отметить, что существует риск: позволяя контрактам потребителей управлять спецификацией поставщика сервиса, мы можем подорвать концептуальную целостность сервиса. Сервисы заключают в себе дискретные, идентифицируемые, переиспользуемые функции, чья целостность не должна быть скомпрометирована необоснованными требованиями, выходящими за пределы их назначения.

## Ссылки

1. [Ian Robinson. Consumer-Driven Contracts](https://martinfowler.com/articles/consumerDrivenContracts.html) (оригинальная статья)
2. [Martin Fowler. Microservices](https://martinfowler.com/articles/microservices.html), раздел Decentralized Governance
3. [Хэм Фокке. Пирамида тестов на практике](https://habr.com/post/358950/), раздел "Контрактные тесты"
4. [Химион Дмитрий. Ядро автоматизации тестирования в микросервисной архитектуре](https://habr.com/company/avito/blog/333644/)

**Метки:**

* [микросервисы](https://habr.com/search/?q=%5B%D0%BC%D0%B8%D0%BA%D1%80%D0%BE%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D1%8B%5D&target_type=posts)
* [микросервисная архитектура](https://habr.com/search/?q=%5B%D0%BC%D0%B8%D0%BA%D1%80%D0%BE%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D0%BD%D0%B0%D1%8F%20%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0%5D&target_type=posts)
* [microservice architecture](https://habr.com/search/?q=%5Bmicroservice%20architecture%5D&target_type=posts)
* [microservices](https://habr.com/search/?q=%5Bmicroservices%5D&target_type=posts)
* [Consumer-Driven Contracts](https://habr.com/search/?q=%5BConsumer-Driven%20Contracts%5D&target_type=posts)
* [архитектура](https://habr.com/search/?q=%5B%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0%5D&target_type=posts)
* [SOA](https://habr.com/search/?q=%5BSOA%5D&target_type=posts)
* [Фаулер](https://habr.com/search/?q=%5B%D0%A4%D0%B0%D1%83%D0%BB%D0%B5%D1%80%5D&target_type=posts)
* [Fowler](https://habr.com/search/?q=%5BFowler%5D&target_type=posts)
* [Jan Robinson](https://habr.com/search/?q=%5BJan%20Robinson%5D&target_type=posts)
* [Ян Робинсон](https://habr.com/search/?q=%5B%D0%AF%D0%BD%20%D0%A0%D0%BE%D0%B1%D0%B8%D0%BD%D1%81%D0%BE%D0%BD%5D&target_type=posts)