8. Сервис-ориентированная архитектура

8.1 Концепция СОА

По определению организации OASIS (Organization for the Advancement of Structured Information Standards), *сервис-ориентированная архитектура (Service-Oriented Architecture – SOA)* – это парадигма организации и использования распределенных возможностей, которые могут принадлежать различным владельцам [56]. В основе сервис-ориентированной архитектуры лежит сущность «действия» (в противоположность сущности «объекта» объектно-ориентированной архитектуры). Типичными составляющими сервис ориентированной архитектуры являются: *сервисные компоненты* (сервисы); *контракты сервисов* (интерфейсы); *соединители сервисов* (транспорт) и *механизмы обнаружения сервисов* (регистры) [53].

Сервисные компоненты (или сервисы) описываются программными компонентами, обеспечивающими прозрачную сетевую адресацию. Существует множество различных определений сервиса. В статье [51] дается следующее определение: сервисами называются открытые, самоопределяющиеся компоненты, поддерживающие быстрое построение распределенных приложений. При этом нет четкого определения, какой объем услуг должен предоставлять отдельный сервис. С одной стороны, *мелкомодульный сервис* может предоставлять элементарный объем функциональной нагрузки и обеспечивать высокую степень повторного использования [63]. В этом случае, для получения желаемого результата, необходимо обеспечить координированную работу нескольких сервисов. С другой стороны, использование *крупномодульных сервисов* позволяет обеспечить хорошую инкапсуляцию функциональности, но затрудняет повторное использование таких сервисов, в связи с их узкой специализацией.

Контракт сервиса (или интерфейс) обеспечивает описание возможностей и качества предоставляемых услуг, предоставляемых конкретным сервисом [51]. В интерфейсе определяется формат сообщений, используемый для обмена информацией, а также входные и выходные параметры методов, поддерживаемых сервисным компонентом. От выбора языка и способа описания интерфейса зависят возможности программной совместимости различных реализаций сервис-ориентированной архитектуры.

Соединитель сервисов (или транспорт) обеспечивает обмен информацией между отдельными сервисными компонентами. Наряду с открытыми стандартами описания интерфейсов, использование гибких транспортных протоколов для обмена информацией между сервисными компонентами позволяет повысить программную совместимость сервис-ориентированной системы [50].

Механизмы обнаружения сервисов (или регистры сервисов) используются для поиска сервисных компонентов, обеспечивающих требуемую функциональность. Среди всего множества различных систем, обеспечивающих обнаружения сервисов [7], можно выделить две основных категории: системы динамического и статического обнаружения. Статические системы обнаружения сервисов (например, UDDI) ориентированы на хранение информации о сервисах в редко изменяющихся системах. Динамические системы обнаружения сервисов [10, 47, 54] ориентированы на системы, в которых допустимо частое появление и исчезновение сервисных компонентов.

На сегодняшний день, веб-сервисы – это наиболее часто встречающейся реализация сервис-ориентированной архитектуры. Веб-сервисы – это технология, разработанная для поддержки взаимодействия между распределенными системами посредством вычислительной сети [72]. В [14] дается следующее определение *веб-сервисов* – это слабосвязанные программные компоненты, поддерживающие многократное использование, которые семантически инкапсулируют отдельные функциональные возможности и программным образом доступны по стандартным протоколам Интернета. Веб-сервисы – это независимая от платформы и языка программирования среда, так как базируются на стандарте XML.

Большинство веб-сервисов используют HTTP для передачи сообщений. Это дает значительное преимущество при разработке распределенных приложений в масштабе Интернет, так как обычно брандмауэры и прокси-серверы без проблем пропускают HTTP-трафик, и в процессе взаимодействия не возникает неожиданных трудностей (которые могут возникнуть, например, при использовании технологии CORBA).

8.2 Связанность программных систем

Связанностью называют степень знания и зависимости одного объекта от внутреннего содержания другого. В соответствии с данным определением, программные системы можно разделить на 2 типа:

- *сильносвязанные системы* (Strong coupling): Java RMI, .NET Remoting и т.п;
- *слабосвязанные системы* (Loose coupling): COA.

Сильная связанность возникает, когда зависимый класс содержит ссылку непосредственно *на определенный класс*, предоставляющий некоторые возможности. В противоположность этому, слабая связанность возникает, когда зависимый класс содержит *ссылку на интерфейс* который может быть реализован одним или несколькими конкретными классами.

Основная цель использования концепции слабосвязанных программных систем – это уменьшение количества зависимостей между компонентами. При уменьшении количества связей, уменьшается объем возможных последствий, возникающих в связи со сбоями или системными изменениями. Наиболее яркие характеристики слабосвязанных распределенных систем приведены в таблице 1 [41].

Таблица 1. Сравнение слабосвязанных и сильносвязанных систем

	Сильносвязанные системы	**Слабосвязанные системы**
Физические соединения	Точка-точка	Через посредника
Стиль взаимодействий	Синхронные	Асинхронные
Модель данных	Общие сложные типы	Простые типы
Связывание	Статическое	Динамическое
Платформа	Сильная зависимость от базовой платформы	Независимость от платформы
Развертывание	Одновременное	Постепенное

Традиционный подход разработки распределенных приложений, поддерживаемый технологиями распределенных объектов, основывается на тесной связи между всеми программными компонентами. Слабосвязанность программных компонентов, поддерживаемая технологией веб-сервисов, позволяет значительно упростить координацию распределенных систем и их реконфигурацию [14].

8.3 Принципы построения COA

Способность двух или более информационных систем (или их компонентов) к взаимодействию, с целью решения определенной задачи и получения определенной информации, называют *интероперабельностью*. Это определение объединяет в себе два понятия:

– *техническая интероперабельность* означает совместимость систем на техническом уровне, включая протоколы передачи данных и форматы их представления;
– *семантическая интероперабельность* — свойство информационных систем, обеспечивающее взаимную употребимость полученной информации на основе общего понимания системами ее значения.

Примером семантической интероперабельности программных систем может служить процесс передачи определенных данных в текстовом виде по каналам связи. Например, если системы семантически не интероперабельны, то получатель не сможет однозначно интерпретировать полученную строку «1.23»: это может быть число с плавающей запятой записанное в десятичной или шестнадцатеричной системе счисления, а может быть дата, которую надо интерпретировать «23 января».

СОА не предписывает жесткой вертикальной («сверху вниз») методологии проектирования, внедрения или управления ИТ-инфраструктурой. Вместо этого, СОА ограничивается лишь рядом принципов, характеризующих каждый из этих процессов; поэтому ее иногда называют не архитектурой, а архитектурным стилем. Отметим некоторые из этих принципов.

– *Распределенное проектирование.* Решения относительно внутренних особенностей информационных систем принимаются различными группами людей, имеющими собственные организационные, политические и экономические мотивы.
– *Постоянство изменений.* Отдельные участки архитектуры могут претерпевать изменения в любой момент времени.
– *Последовательное совершенствование.* Локальное улучшение компонентов архитектуры должно приводить к совершенствованию всей архитектуры в целом – к росту суммарной полезности компонентов того же уровня, что и изменяемый, равно как и компонентов более низкого и более высокого уровня. Например, известный веб-сервис Google Translate постоянно претерпевает изменения. Изначально, он обеспечивал только веб-интерфейс для перевода и ограниченный набор языков. Постепенно увеличивались функциональные возможности сервиса: расширялся набор языков, появилась возможность голосового воспроизведения перевода, при переводе отдельного слова начали выдаваться словарные статьи с несколькими результатами перевода и т.п. При этом API и интерфейс менялся настолько незначительно, что клиенты могли использовать новые возмож-

ности посредством старого API (например, получение словарных статей) или же не менять используемый интерфейс до тех пор, пока не потребуется использование новых возможностей.

– *Рекурсивность.* Однотипные решения имеют место на различных уровнях архитектуры.

8.4 Подход СОА

С точки зрения информационных технологий, логика предприятия может быть разделена на две важных половины: бизнес-логика и логика приложения (рис. 23). Каждый слой существует в своем собственном «мире».

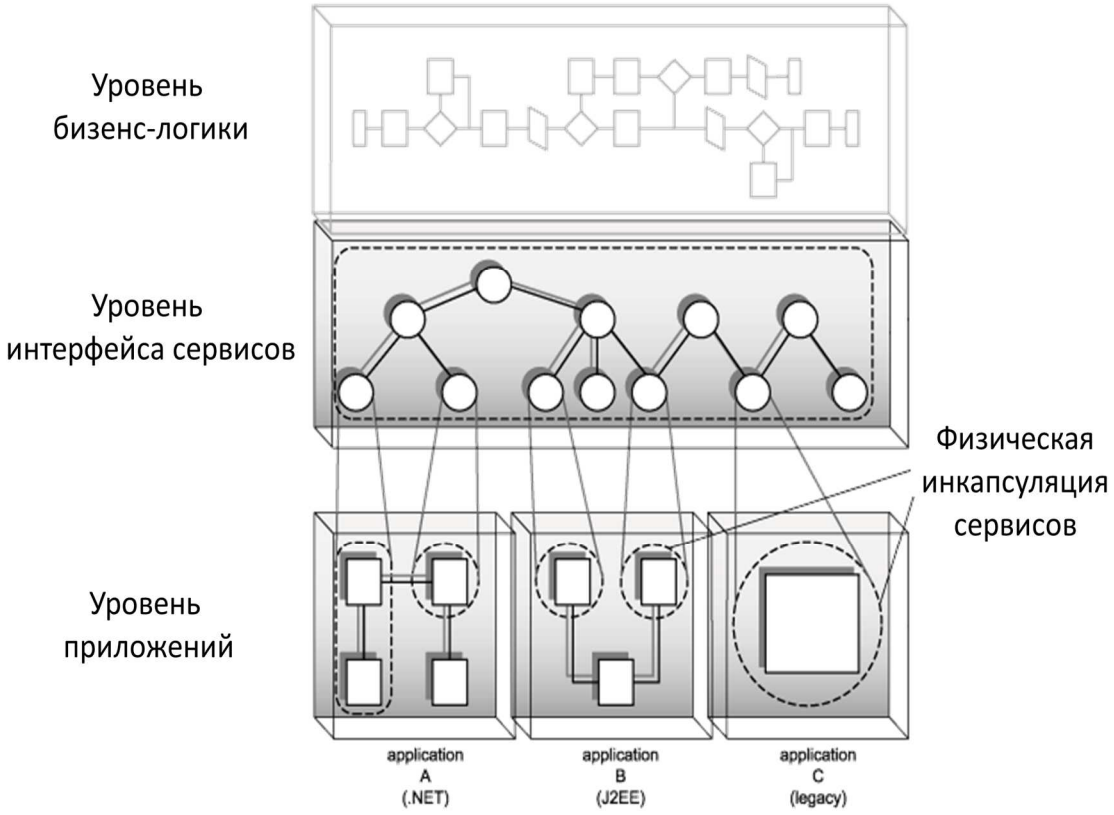

Рис. 23. Уровни логики предприятия

Бизнес-логика является документальной реализацией бизнес-требований, которые исходят из проблемной области, в которой работает предприятие. Бизнес-логика, как правило, структурирована в процессах, которые выражают эти требования, а также ограничениях и зависимости от внешних влияний.

Логика приложения – это реализация бизнес-логики, организованная на основе различных технологических решений. Логика приложения выражает процессы бизнес-логики посредством приобретенных или специально разработанных программных систем в условиях ограниченных технических возможностей и зависимостей от поставщика решения.

Процесс преобразования бизнес-логики в логику приложений и реализация сервисов на основе данных требований является процессом создания сервисно-ориентированной инфраструктуры для задач предприятия. Не существует «догматических» принципов построения СОА, но при реализации собственной инфраструктуры желательно придерживаться некоторых основных подходов, описанных ниже.

1. Сервисы должны поддерживать повторное использование. СОА-системы должны поддерживать повторное использование всех сервисов, независимо от сиюминутных требований к их функциональным особенностям. Если при разработке системы постараться максимально учесть это требование, то повышаются шансы значительно упростить процесс решения задач, которые непременно появятся в будущем, при развитии системы. Также изначально ориентированный на повторное использование сервис позволяет избежать разработки «обертки», которая бы подстраивала старый сервис для решения новых задач.

Так как сервис – это не что иное, как просто набор связных операций, логика каждой индивидуальной операции, предоставляемой сервисом, должна поддерживать повторное использование.

Например, в компании A-comp для отправки счета в систему биллинга «B-comp Account Payable Service» используется метод `SubmitInvoice`. Для того чтобы проверить состояние счета, необходимо постоянно проверять метаданные отправленного счета. Для этого используется метод `GetBcompMetadata`.

Т.к. эти операции ориентированы на решение *сиюминутных*, и вполне конкретных задач, они *не имеют потенциала повторного использования*. Метод `SubmitInvoice` разработан таким образом, чтобы транслировать сообщение в специфичном XML-формате, соответствующем системе B-comp, то есть «заточен» на конкретную версию протокола обмена информацией и конкретные данные. Метод `GetBcompMetadata` (как мы можем понять из имени метода) изначально ориентирован на использование исключительно с компанией B-comp, и не допускает повторного использования с любой другой системой биллинге.

Если бы компания A-comp задумалась о правильной сервисно-ориентированной инфраструктуре, то были бы разработан универсальный интерфейс с методами типа «`SubmitInvoice`» и «`CheckInvoice`», реализацию которых можно было бы использовать для любой внешней системы биллинга.

2. Сервисы должны обеспечивать формальный контракт использования. Контракт сервиса предоставляет следующую информацию:

- конечную точку (service endpoint): адрес, по которому можно обратиться к данному сервису;
- все операции, предоставляемые сервисом;
- все сообщения, поддерживаемые каждой операцией;
- правила и характеристики сервиса и его операций.

3. Сервисы должны быть слабосвязаны. Никто не может предугадать, в какую сторону будет развиваться IT-инфраструктура. Решения могут развиваться, взаимодействовать, заменять друг друга. В связи с этим основной задачей является сохранение целостности системы в рамках такого развития, независимо от происходящих изменений.

Система сервисов является слабосвязанной, если сервис может приобретать знания о другом сервисе, оставаясь независимым от внутренней реализации логики данного сервиса. Это достигается посредством использования контрактов сервисов.

Слабосвязанность программных компонентов, лежащая в основе СОА, позволяет значительно упростить координацию распределенных систем и их реконфигурацию. Основная цель использования концепции слабосвязанных программных систем – это уменьшение количества зависимостей между компонентами. При уменьшении количества связей, уменьшается объем возможных последствий, возникающих в связи со сбоями или системными изменениями.

Слабосвязанность – это не синоним «инкапсуляции» объектно-ориентированной концепции построения программных систем. Программная система может полностью соответствовать требованиям инкапсуляции, но быть сильносвязанной на семантическом уровне.

Например, существует сервис, который должен передавать большой объем текстовой информации в открытом виде другому сервису. При этом символ новой строки передавать нельзя, так как это неадекватно обрабатывается XML-парсером, и вся информация после знака новой строки теряется. Что же делать в этом случае? Первое решение, которое приходит в голову разработчику – это замена всех вхождений символов переноса строки на любой другой редко встречающийся в передаваемых сообщениях символ (или последовательность символов). В этом случае единственная деталь, которую необходимо учесть –

это описать в приемнике простейшую обратную процедуру, которая будет делать обратную замену, и вопрос решен. Поступая так, разработчик игнорирует принцип слабосвязанности программных систем, так как *в этом случае клиент и сервер будут взаимодействовать между собой не на основе открытого интерфейса, а на основе информации о внутренних процессах работы друг друга.*

Разработчик не задумывается о последствиях, которые может повлечь за собой в дальнейшем такое решение. Если через несколько месяцев (когда этот «финт» забудется) сторонний разработчик захочет создать собственного клиента для данного сервиса он будет неприятно удивлен, увидев вместо ожидаемого форматированного текста странные символы. Еще большая проблема возникнет, если вдруг в самом тексте будет появляться последовательность символов, на которую разработчик заменил символ новой строки. Поддержка такой системы превратится в постоянную пытку.

Существует несколько вариантов решения данной задачи посредством перехода к концепции слабосвязанных систем. Одним из оптимальных решений является переход от передачи данных в виде строки к передаче данных в виде массива строк. Это будет явно отражено в интерфейсе системы, соответственно, любой клиент будет знать, что данные к нему будут переданы в виде массива строк, каждая из которых заканчивается символом перехода на новую строку.

4. Сервисы должны абстрагировать внутреннюю логику. Каждый сервис должен действовать как «черный ящик», скрывающий свои детали от окружающего мира. Нет четкого определения, какой объем логики должен помещаться в отдельном сервисе. Взаимодействие на уровне интерфейсов является одним из требований для обеспечения слабой связанности.

5. Сервисы должны быть совместимы. Сервис может как самостоятельно реализовывать логику, так и применять другие сервисы для ее реализации. Сервисы должны быть спроектированы таким образом, чтобы поддерживать возможность их использования в качестве элементов другого сервиса. Принцип совместимости не зависит от того, использует ли сервис для выполнения своей работы другие сервисы.

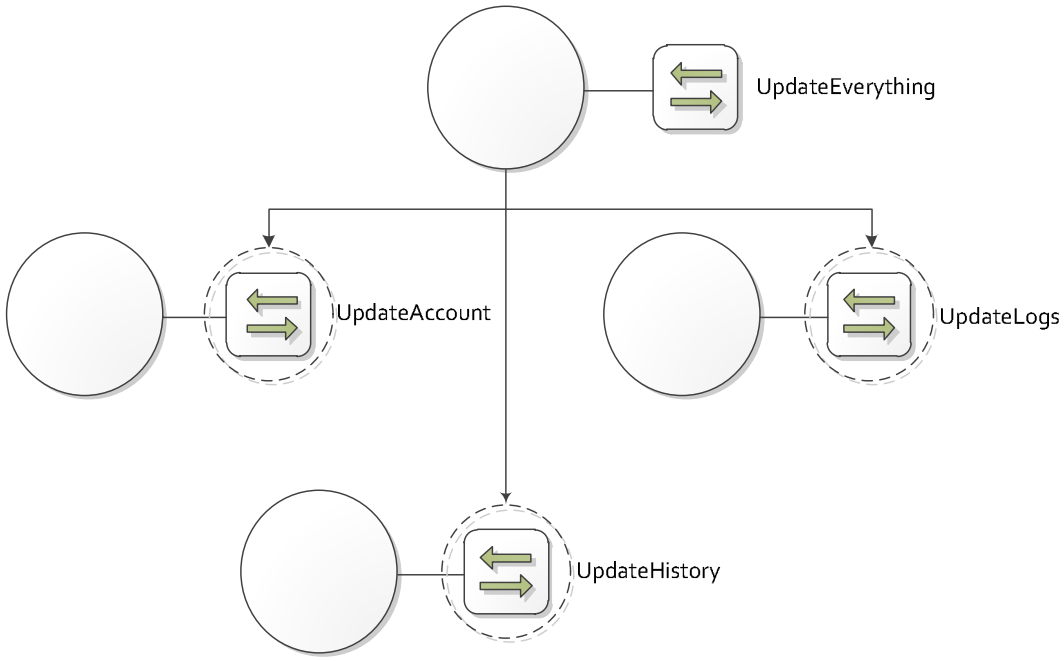

Рис. 24. Сервисы, используемые в качестве элементов другого сервиса

В качестве примера такого взаимодействия сервисов выступает концепция оркестрации сервисов. В этом случае, сервис-ориентированный процесс (который в принципе может быть определен как композиция сервисов) управляется сервисом родительского процесса, который включает в себя другие сервисы, являющиеся участниками данного процесса.

Кроме того, принцип совместимости также определяет вид сервисных операций. Совместимость – это, по сути, просто другая форма повторного использования, и поэтому операции должны быть стандартными, а для наибольшей совместимости должны обладать необходимым уровнем детализации.

6. Сервисы должны быть автономными. Свойство автономности требует, чтобы область бизнес-логики и ресурсов, используемых сервисом были ограничены явными пределами. Это позволяет сервису самому управлять всеми своими процессами (рис. 25). Также это устраняет зависимость от других сервисов, что освобождает сервис от связей, которые могут препятствовать его применению и развитию. Вопрос автономности – наиболее важный аргумент при распределении бизнес-логики на отдельные сервисы.

Обратите внимание, что автономность не обязательно предоставляет сервису исключительное право собственности на бизнес-логику, которую он инкапсулирует. Есть только гарантия того, что во время исполнения сервис контролирует любую логику, которую он реализует. Поэтому мы можем выделить два типами автономности.

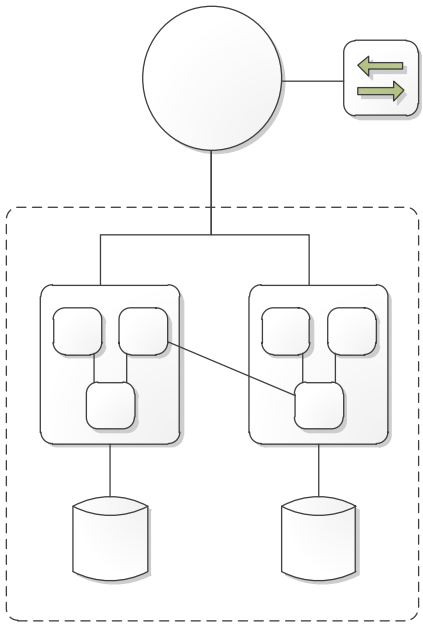

Рис. 25. Автономность сервиса: во время выполнения сервис управляет логикой нижнего уровня

– *Автономность на уровне сервиса*: границы ответственности сервисов отделены, но они могут использовать общие ресурсы. Например, сервис обертки, который инкапсулирует унаследованную программную систему, которая также кем-то используется (независимо от данного сервиса), обладает автономностью данного типа. Он управляет унаследованной системой, но также совместно использует ресурсы с другими существующими клиентами.
– *Чистая автономность:* бизнес-логика и ресурсы находятся под полным контролем сервиса. Как правило, такой вид автономности используется, когда для реализации сервиса бизнес-логика создается с нуля.

7. Сервисы не должны использовать информацию о состоянии. Сервисы должны сводить к минимуму объем информации о состоянии, и время, в течение которого они ею обладают. Информация о состоянии – это определенные данные, характеризующие текущую деятельность. Например, пока сервис обрабатывает сообщение, он временно зависит от состояния (stateful). Если сервис несет ответственность за сохранение состояния в течение более длительного времени, его способность оставаться доступным для других клиентов будет затруднена.

Независимость от состояния (statelessness) позволяет повысить возможности масштабируемости и повторного использования сервисов. Чтобы сервис

как можно меньше зависел от состояния, его операции должны быть разработаны с учетом соображений обработки информации без данных о состоянии.

Основной особенностью COA, поддерживающей независимость от состояния, является использование сообщений-документов. Чем выше сложность сообщения, тем более независимым и самодостаточным оно остается.

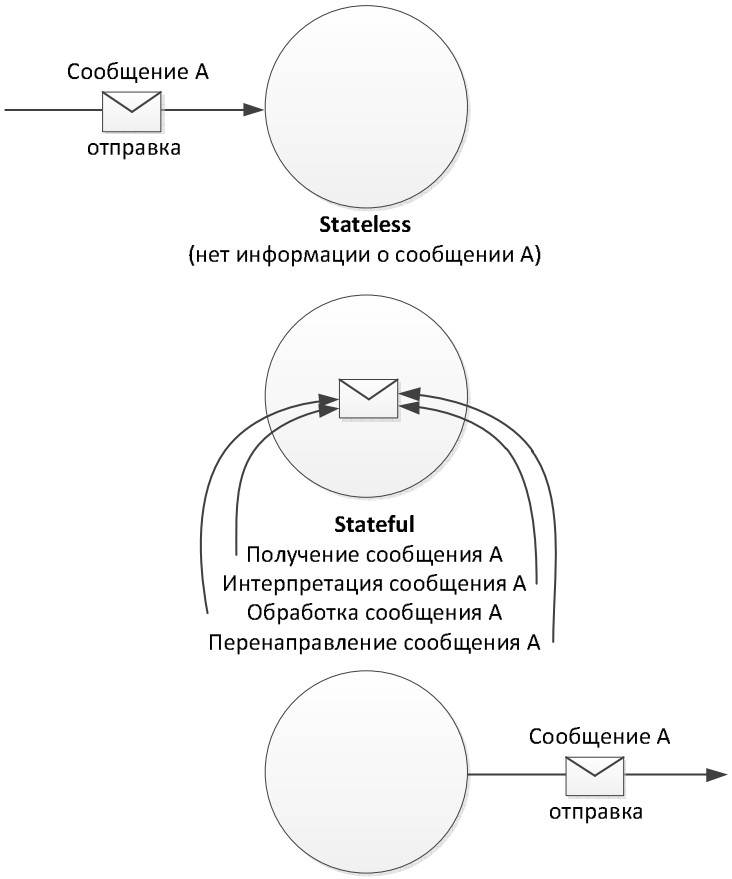

Рис. 26. Временная зависимость от состояния во время обработки сообщения

8. Сервисы должны поддерживать обнаружение. Обнаружение сервисов позволяет избежать случайного создания избыточного сервиса, обеспечивающего избыточную логику. Метаданные сервиса должны подробно описать не только общую цель сервиса, но и функциональность, реализуемую его операциями.

На уровне COA, обнаружение характеризует способность архитектуры обеспечить механизмы поиска, такие как реестр или каталог. На уровне сервиса, принцип обнаружения относится к процессу проектирования отдельного сервиса, так чтобы данный сервис настолько подавался обнаружению, насколько это возможно.

Рассмотрим пример. Компания RailCo не обеспечивает никаких средств обнаружения своих сервисов, как внутри своей компании, так и для внешних

пользователей. Хотя каждый сервис и обладает собственным WSDL-документом и в полной мере способен выступать в качестве поставщика услуг, сервис подачи счета в первую очередь используется в качестве элемента другого сервиса и не ожидает никаких сообщений от других сервисов, кроме сервиса оплаты счетов.

Сервис выполнения заказа (RailCo) был вручную зарегистрирован с помощью B2B-решения компании TLS, так что он будет помещен в список доступных сервисов. Этот сервис не предоставляет функциональность многоразового использования, и поэтому считается, что он не обладает способностью к обнаружению.

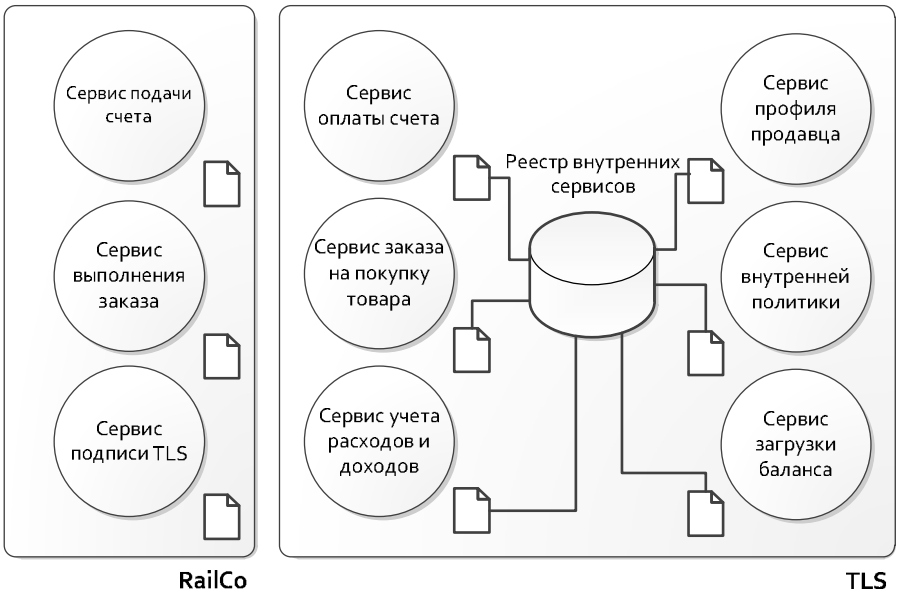

Рис. 27. Обнаружение на уровне логики предприятия

Из-за многоразового характера использования сервисов компании TLS и из-за количества сервисов, которые, как ожидалось, будут реализованы в компании TLS, был создан реестр внутренних сервисов (рис. 27). Эта часть инфраструктуры TLS способствует обнаружению сервисов и предотвращению случайной избыточности.