Лекция 6

Сервис-ориентированная архитектура

Сервис-ориентированная архитектура *(COA, Service-Oriented Architecture – SOA)* – это парадигма организации и использования распределенных возможностей, которые могут принадлежать различным владельцам.

COA – это модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.

Типичные составляющие COA:

- *сервисные компоненты* (сервисы);
- *контракты сервисов* (интерфейсы);
- *соединители сервисов* (транспорт);
- *механизмы обнаружения сервисов* (регистры).

Сервисные компоненты (или сервисы) – это открытые, самоопределяющиеся программные компоненты, предоставляющие определенную функциональность.

В зависимости от объема предоставляемых услуг выделяют сервисы:

- *мелкомодульные (ММС),* предоставляющие элементарный объем функциональной нагрузки и обеспечивающие высокую степень повторного использования; иногда для получения желаемого результата, необходимо обеспечить координированную работу нескольких ММС;
- *крупномодульные (КМС),* позволяющие обеспечить хорошую инкапсуляцию функциональности; повторное использование КМС затруднено в связи с их узкой специализацией.

Контракт сервиса (или интерфейс) обеспечивает описание возможностей и качества предоставляемых услуг, предоставляемых конкретным сервисом. 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 СОА.

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.

Механизмы обнаружения сервисов (или регистры сервисов) используются для поиска сервисных компонентов, обеспечивающих требуемую функциональность.

Выделяют 2 основных категории систем обнаружения:

- *статические системы* обнаружения сервисов (например, UDDI) ориентированы на хранение информации о сервисах в редко изменяющихся системах;
- *динамические системы* обнаружения сервисов ориентированы на системы, в которых допустимо частое появление и исчезновение сервисных компонентов.

Связанность программных систем

Связанностью называют степень знания и зависимости одного объекта от внутреннего содержания другого.

Программные системы можно разделить на 2 типа:

- *сильносвязанные системы (Strong coupling):* зависимый класс содержит ссылку непосредственно *на определенный класс,* предоставляющий некоторые возможности (примеры: Java RMI, NET Remoting);
- *слабосвязанные системы (Loose coupling):* зависимый класс содержит *ссылку на интерфейс* который может быть реализован одним или несколькими конкретными классами (пример: COA).

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.

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.

Сравнение сильносвязанных и слабосвязанных 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

Принципы построения СОА

Интероперабельность – способность двух или более информационных систем (или их компонентов) к взаимодействию, с целью решения определенной задачи и получения определенной информации.

Это определение объединяет в себе два понятия:

- *техническая интероперабельность* – 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. СОА ограничивается рядом принципов, характеризующих каждый из этих процессов; поэтому ее иногда называют не архитектурой, а архитектурным стилем.

Основные принципы построения СОА:

⇨ ***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 (Application Programming Interface) и интерфейс менялся незначительно.

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

Подход СОА

С точки зрения информационных технологий, ***логика предприятия*** может быть разделена на: *бизнес-логику (БЛ)* и *логику приложения (ЛП)*.

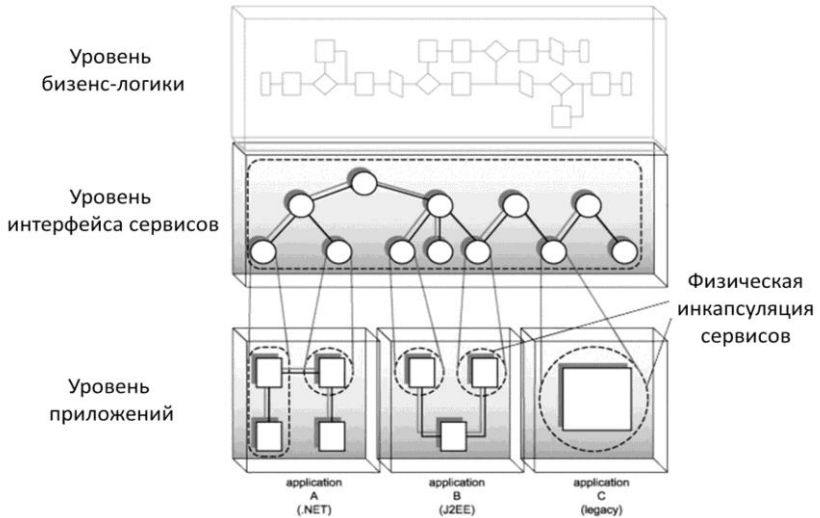

Рисунок 15 – Уровни логики предприятия

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

Логика приложения – это реализация БЛ, организованная на основе различных технологических решений. ЛП выражает процессы БЛ 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.**

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.

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.

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.

Совместимость – это, по сути, просто другая форма повторного использования, и поэтому операции должны быть стандартными, а для

наибольшей совместимости должны обладать необходимым уровнем детализации.

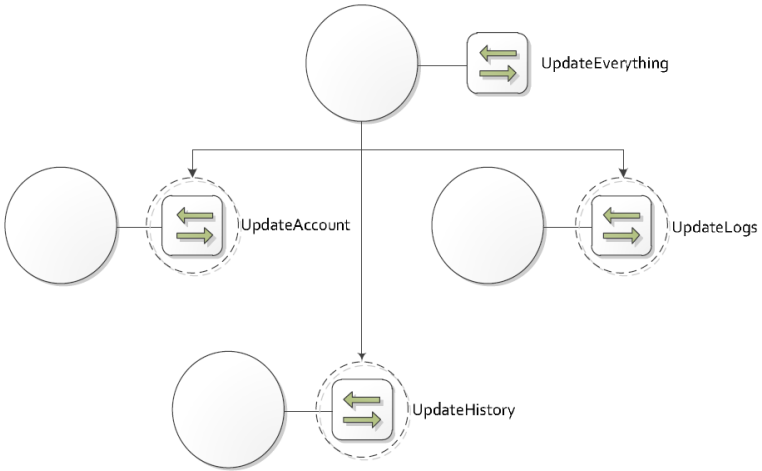

Рисунок 16 – Сервисы, используемые в качестве элементов другого сервиса

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. 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.

Автономность не обязательно предоставляет сервису исключительное право собственности на бизнес-логику, которую он инкапсулирует.

Есть 2 типа автономности:

- *Автономность на уровне сервиса:* границы ответственности сервисов отделены, но они могут использовать общие ресурсы.
- *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.

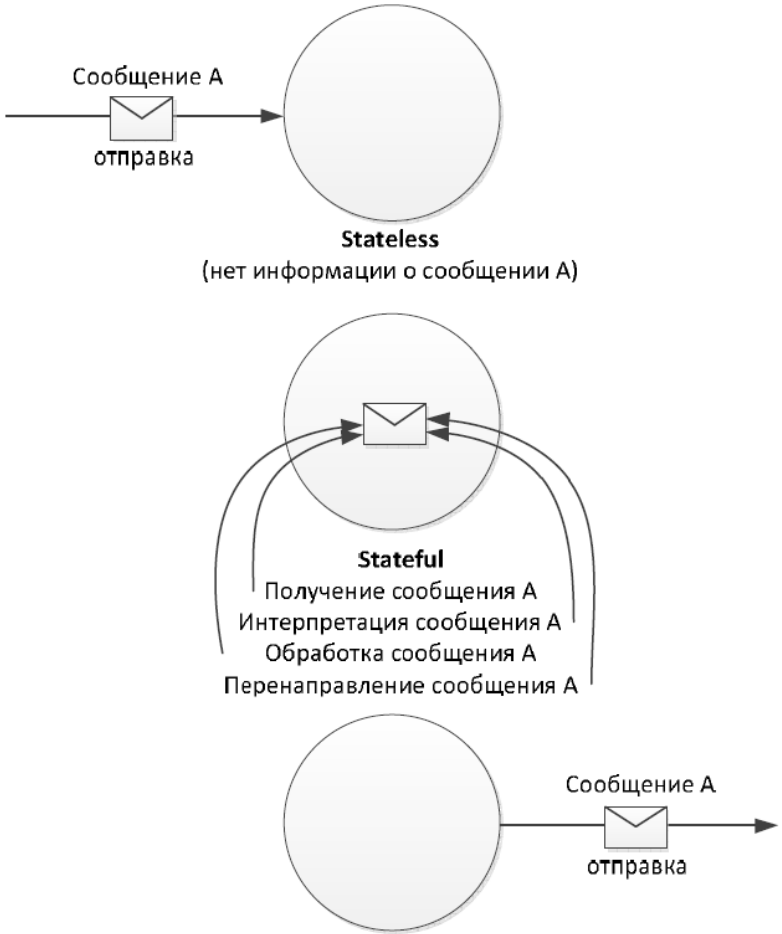

Рисунок 17 – Временная зависимость от состояния во время обработки сообщения

Независимость от состояния (statelessness) позволяет повысить возможности масштабируемости и повторного использования сервисов. Операции сервиса должны быть разработаны с учетом соображений обработки информации без данных о состоянии.

Для поддержки независимости от состояния в СОА используются *сообщения-документы*. Чем сложнее сообщение, тем более независимым и самодостаточным оно остается.

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 СОА, 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.

Рисунок 18 – Свойства сервисов согласно подходу СОА