МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

Архитектура программного обеспечения инфокоммуникационных систем

Учебное пособие

Автор: Осипов Никита Алексеевич

Санкт-Петербург

2013

СОДЕРЖАНИЕ

Введение ... 3
1. Архитектура и дизайн программного обеспечения ... 4
1.1 Понятие архитектуры программного обеспечения ... 4
1.2 Основные принципы проектирования архитектуры ПО ... 14
1.3 Архитектурные шаблоны и стили ... 23
1.4 Методика построения архитектуры и дизайна ... 41
2. Основы проектирования многослойных приложений ... 55
2.1 Логическое представление архитектуры многослойной системы ... 55
2.2 Этапы проектирования многослойной структуры ... 58
2.3 Общие принципы проектирования ... 64
2.4 Шаблоны проектирования ... 73
2.5 Проектирование компонентов приложения ... 81
2.6 Проектирование компонентов пользовательского интерфейса и компонентов логики представления ... 87
2. 7 Проектирование компонентов бизнес - слоя ... 100
2.8 Проектирование компонентов данных ... 107
2.9 Показатели качества ... 115
2.10 Проектирование общей функциональности приложения ... 118
2.11 Стратегии развертывания приложений ... 120
3. Архитектура основных типов приложений ... 132
3.1 Типы приложений ... 133
3.2 Принципы проектирования и основные атрибуты Веб-приложения ... 141
3.3 Проектирование насыщенных клиентских приложений ... 143
3.4 Проектирование насыщенных Веб- приложений ... 145
3.5 Проектирование мобильных приложений ... 147
3.6 Проектирование сервисных приложений ... 150
Заключение ... 153
Литература ... 154

Введение

Как и любая другая сложная структура, программное обеспечение (ПО) должно строиться на прочном фундаменте. Неправильное определение ключевых сценариев, неправильное проектирование общих вопросов или неспособность выявить долгосрочные последствия основных решений могут поставить под угрозу все приложение.

В середине 90-х годов прошлого века благодаря широкому распространению клиент-серверного подхода и его трансформации в “многозвенный клиент-сервер” вопросы организации архитектуры программного обеспечения стали складываться в самостоятельную и достаточно обширную дисциплину. В результате, сформировалась точка зрения на архитектуру не только в приложении к конкретной программной системе, но и развился взгляд на архитектуру, как на приложение общих принципов организации программных компонент. В итоге, уже на сегодняшний день, на фоне такого развития понимания архитектуры, накоплен целый комплекс подходов и созданы различные архитектурные комплексы методов и инструментов, призванные в той или иной степени формализовать имеющийся в индустрии опыт [1-3,4,6,10,11,13,16,18].

С одной стороны современные инструменты и платформы упрощают задачу по созданию приложений, а с другой не устраняют необходимость в тщательном их проектировании на основании конкретных сценариев и требований.

Очевидно, что неправильно выработанная архитектура обусловливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации.

Данное учебное пособие позволит понять базовые принципы и шаблоны построения архитектуры и дизайна для разработки успешных программных решений, правильно выбрать стратегии и шаблоны проектирования, которые помогут при проектировании слоев, компонентов и сервисов решения, а также правильно выбрать технологии для реализации системы и создать возможный вариант базовой архитектуры.

В первом разделе дается обзор базовых принципов и шаблонов, которые являются основой для создания хорошей архитектуры и дизайна приложений, и предлагается подход к созданию дизайна приложения.

Во втором разделе даются рекомендации по проектированию слоев, компонентов и сервисов решения, а также по реализации показателей качества.

В третьем разделе описывается проектирование архитектуры общих типов приложений, таких как Веб-приложения, насыщенные Веб-приложения (Rich Internet Applications –RIA), насыщенные клиентские приложения.

1. Архитектура и дизайн программного обеспечения

1.1 Понятие архитектуры программного обеспечения

Создание архитектуры приложения – это процесс формирования структурированного решения, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества, такие как производительность, безопасность и управляемость.

Принятое решение может иметь существенное влияние на качество, производительность, удобство обслуживания и общий успех приложения.

В литературе встречается несколько вариантов определения понятия архитектуры программного обеспечения. Все они в общем смысле похожи, но отличаются своими особенностями, по-своему реализующие построение архитектуры.

Архитектура программной системы [10] – это структура системы, включающая программные элементы, видимые извне свойства этих элементов и взаимоотношения между ними. Авторы подчеркивают, что архитектура касается внешней части интерфейсов, а внутренние детали элементов – детали, относящиеся исключительно к внутренней реализации не являются архитектурными.

Стандарт IEEE 1471 определяет архитектуру и связанные с ней понятия следующим образом:

Архитектура – это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.

Система – это набор компонентов, объединенных для выполнения определенной функции или набора функций. Термин "система" охватывает отдельные приложения, системы в традиционном смысле, подсистемы, системы систем, линейки продуктов, семейства продуктов, целые корпорации и другие агрегации, имеющие отношение к данной теме. Система существует для выполнения одной или более миссий в своем окружении.

Окружение, или контекст, определяет ход и обстоятельства экономических, эксплуатационных, политических и других влияний на систему.

Миссия – это применение или действие, для которого одно или несколько заинтересованных лиц планируют использовать систему в соответствии с некоторым набором условий.

Заинтересованное лицо – это физическое лицо, группа или организация (или ее категории), которые заинтересованы в системе или имеют связанные с ней задачи.

В [11] предложена следующая формулировка:

Программная архитектура представляет собой набор существенных решений, касающихся организации программной системы, таких как:

- выбор структурных элементов и их интерфейсов, из которых слагается система;
- поведение, определяемое взаимодействием этих элементов;

- объединение этих структурных и функциональных элементов в более крупные подсистемы;
- архитектурный стиль, исповедуемый в данной организации.

Рассматривая указанные формулировки архитектуры можно отметить следующие особенности, присущие этому процессу:

- разделение системы на составные части на начальном этапе проектирования;
- необходимость в принятии решений, которые трудно изменить впоследствии;
- выработка множества возможных вариантов архитектуры для системы;
- важность с точки зрения архитектуры различных аспектов может меняться в процессе жизненного цикла системы;

Таким образом, архитектура программного обеспечения заключает в себе ряд важных решений об организации программной системы, среди которых следует отметить следующие:

- выбор структурных элементов и их интерфейсов, составляющих и объединяющих систему в единое целое;
- поведение, обеспечиваемое совместной работой этих элементов;
- организацию этих структурных и поведенческих элементов в более крупные подсистемы,
- архитектурный стиль, которого придерживается конкретная организация.

Выбор архитектуры ПО также касается функциональности, удобства использования, устойчивости, производительности, повторного использования, понятности, экономических и технологических ограничений, эстетического восприятия и поиска компромиссов.

Значение архитектуры

Во введении был озвучен тезис о том, что такая сложная структура, как программное обеспечение должно строиться на прочном фундаменте. Неправильное определение ключевых сценариев, неправильное проектирование общих вопросов или неспособность выявить долгосрочные последствия основных решений могут поставить под угрозу все приложение. Современные инструменты и платформы упрощают задачу по созданию приложений, но не устраняют необходимости в тщательном их проектировании на основании конкретных сценариев и требований. Неправильно выработанная архитектура обусловливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации.

В связи с этим, следует говорить не просто об архитектуре, а о хорошей, правильной или даже «красивой» архитектуре. И это притом, что в процессе своей работы архитектору постоянно приходится принимать компромиссные решения, чтобы учесть предъявляемые к системе требования, и, очевидно, что для заданного набора функциональных и качественных требований не существует единственно правильной архитектуры, единственного верного

решения. Перед тем как начинать построение, тестирование и развертывание системы, необходимо оценить архитектуру и определить, удовлетворяет ли она поставленным требованиям. В конечном счете, чтобы построить архитектуру необходимо основываться на неких критериях, которые отделили бы хорошую архитектуру от плохой. Этому важному вопросу в дальнейшем (см. п. 2.9) будет уделено особое внимание. В области программных продуктов существует ресурс [13], в котором собраны образцовые продукты, так называемый «зал славы» - Software Product Line Hall of Fame. Ярким примером удачной архитектуры является архитектура WWW (World Wide Web), созданная Тимом Бернерсом-Ли, особенностью которой является то, что изменяющиеся задачи и потребности клиентов отражаются на уровне архитектуры [4,10].

Проектирование систем должно осуществляться с учетом потребностей пользователя, системы (ИТ-инфраструктуры) и бизнес-целей. Для каждой из этих составляющих определяются ключевые сценарии и выделяются важные параметры качества (например, надежность или масштабируемость), а также основные области удовлетворенности и неудовлетворенности. По возможности необходимо выработать и учесть показатели успешности в каждой из этих областей. Необходимо искать компромиссы и находить баланс между конкурирующими требованиями этих трех областей.

Например, пользовательский интерфейс решения очень часто является производным от бизнес-целей и ИТ-инфраструктуры, и изменения одного из этих компонентов могут существенно влиять на результирующие механизмы взаимодействия с пользователем. Аналогично, изменения в требованиях по взаимодействию с пользователем могут оказывать сильное воздействие на требования к бизнес-области и ИТ-инфраструктуре.

Например, основной целью пользователя и бизнеса может быть производительность, но системный администратор, вероятно, не сможет инвестировать в оборудование, необходимое для реализации этой цели, 100 % своего рабочего времени. Компромиссным решением может быть выполнение данной цели на 80%.

Неправильное определение ключевых сценариев, неправильное проектирование общих вопросов или неспособность выявить долгосрочные последствия основных решений могут поставить под угрозу все приложение. Современные инструменты и платформы упрощают задачу по созданию приложений, но не устраняют необходимости в тщательном их проектировании на основании конкретных сценариев и требований. Неправильно выработанная архитектура обусловливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации.

Основное назначение архитектуры – описание использования или взаимодействия основных элементов и компонентов приложения.

Связь архитектуры и проектирования

Выбор структур данных и алгоритмов их обработки или деталей реализации отдельных компонентов являются вопросами проектирования. Часто вопросы архитектуры и проектирования пересекаются.

Вместо того чтобы вводить жесткие правила, разграничивающие архитектуру и проектирование, имеет смысл комбинировать эти две области. В некоторых случаях, принимаемые решения, очевидно, являются архитектурными по своей природе, в других – больше касаются проектирования и реализации архитектуры.

Функциональность системы не является основной заботой программного архитектора. Архитектор должен сосредоточить свое внимание на качественные требования, которые необходимо удовлетворить. Качественные требования определяют, каким образом должна быть предоставлена функциональность системы, чтобы она оказалась приемлемой для сторон, заинтересованных в успехе системы.

Основные исходные вопросы при разработке архитектуры ПО могут быть следующие:

- как пользователь будет использовать приложение?
- как приложение будет развертываться и обслуживаться при эксплуатации?
- какие требования по атрибутам качества, таким как безопасность, производительность, возможность параллельной обработки, интернационализация и конфигурация, выдвигаются к приложению?
- как спроектировать приложение, чтобы оно оставалось гибким и удобным в обслуживании в течение долгого времени?
- основные архитектурные направления, которые могут оказывать влияние на приложение сейчас или после его развертывания?

Например, при разработке архитектуры веб-приложения архитектор вместо вопросов о макете страниц и деревьях навигации должен задать следующие вопросы:

- кто предоставит хостинг? Существуют ли в среде хостер-провайдера ограничения на применяемые технологии?
- будет ли приложение работать в Windows Server?
- сколько одновременно работающих пользователей должно поддерживать приложение?
- насколько безопасным должно быть приложение? Существуют ли данные, которые необходимо защитить? Будет ли приложение развернуто для общего доступа в Интернете или в приватной интрасети?
- можно ли назначить приоритеты ответам на эти вопросы? Например, можно ли считать, что количество пользователей важнее времени отклика?

В зависимости от ответов на эти вопросы можно переходить к построению эскиза архитектуры системы.

Следует заметить, что, как правило, область использования приложения известна архитектору, и он может предположить, какие решения окажут наиболее заметное влияние на архитектуру.

На основе сформулированных вопросов можно выделяет два важных принципа, типичных для успешных архитекторов [4]: привлечение к работе заинтересованных сторон, и ориентация, как на качественные требования, так и на функциональность. Архитектор должен сначала спросить у клиента, чего он хочет от системы, и каковы приоритеты его потребностей.

Таким образом, архитектура сосредотачивается на следующих конкретных аспектах:

- структура модели - организационные паттерны, например, деление на уровни (layering);
- наиболее важные элементы - критические сценарии использования, основные классы, общие механизмы и т.п., но не все элементы, существующие в модели;
- несколько ключевых сценариев, показывающих основные потоки управления в системе;
- службы, модульность, необязательная функциональность, аспекты, связанные с линейкой продуктов.

Архитектурные представления являются абстрактными, или упрощенными представлениями общего дизайна, где наиболее важные характеристики выделены, а детали скрыты.

Назначение и цели архитектуры

Основное назначение архитектуры – описание использования или взаимодействия основных элементов и компонентов приложения.

Архитектура приложения должна объединять бизнес-требования и технические требования через понимание вариантов использования с последующим нахождением путей их реализации в ПО.

Цель архитектуры – выявить требования, оказывающие влияние на структуру приложения.

Хорошая архитектура:

- снижает бизнес-риски, связанные с созданием технического решения;
- обладает значительной гибкостью, чтобы справляться с естественным развитием технологий, как в области оборудования и ПО, так и пользовательских сценариев и требований.

Архитектор должен учитывать общий эффект от принимаемых проектных решений, обязательно присутствующие компромиссы между атрибутами качества (такими как производительность и безопасность) и компромиссы, необходимые для выполнения пользовательских, системных и бизнес-требований.

Необходимо помнить, что архитектура должна:

- раскрывать структуру системы, но скрывать детали реализации;
- реализовывать все варианты использования и сценарии;

- по возможности отвечать всем требованиям различных заинтересованных сторон;
- выполнять требования по функциональности и по качеству.

Выше было отмечено, что архитектура ПО часто описывается как организация или структура системы, где система представляет набор компонентов, выполняющих определенную функцию или набор функций.

Таким образом, основное назначение архитектуры состоит в организации компонентов с целью обеспечения определенной функциональности. Такую организацию функциональности часто называют группировкой компонентов по «функциональным областям» [6]. На рис. 1.1 представлена типовая архитектура приложения, компоненты которого сгруппированы по функциональным областям.

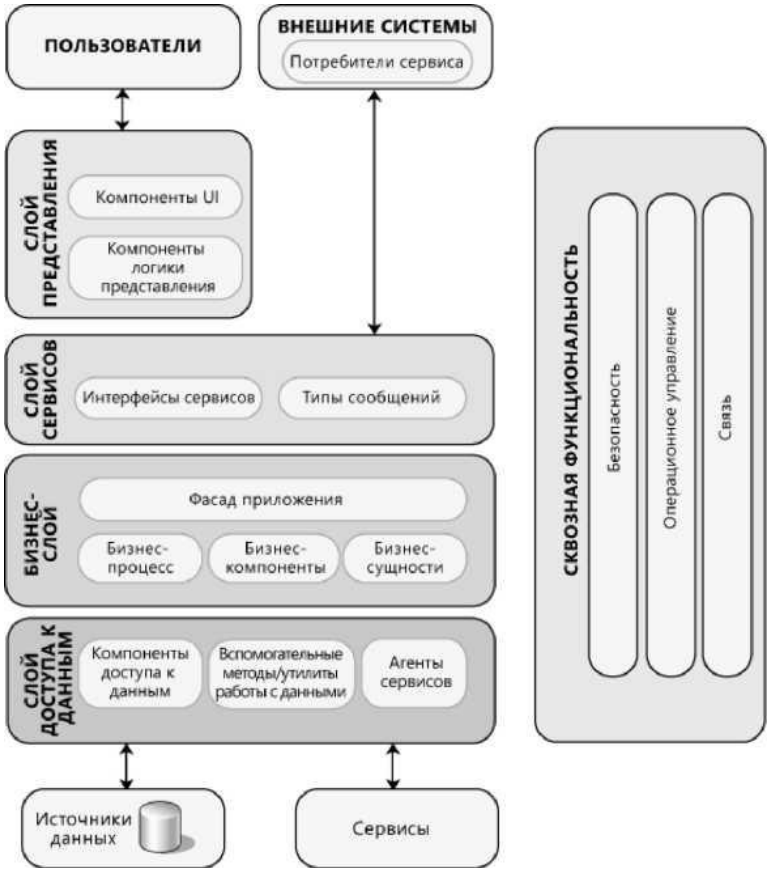

Рис. 1.1 Типовая архитектура приложения

Функциональные области используются не только для группировки компонентов, некоторые из них посвящены взаимодействию и организации совместной работы компонентов. Ниже приводятся рекомендации по различным функциональным областям, которыми необходимо руководствоваться при проектировании архитектуры собственного приложения.

Формирование архитектурных решений[1]

При разработке архитектуры важно учитывать основные факторы, которые формируют архитектурные решения сегодня и будут оказывать влияние на то, как изменятся архитектурные решения в будущем.

[1] Семинар №1.

Эти факторы определяются пожеланиями пользователей, требованиями бизнеса о получении более быстрых результатов, лучшей поддержки изменяющихся стилей работы и рабочих процессов, а также улучшенной адаптируемости дизайна ПО.

Рассмотрим следующие основные направления:

1. Наделение пользователя полномочиями.

Дизайн, поддерживающий наделение пользователя полномочиями, является гибким, настраиваемым и ориентированным на пользователя. Проектируйте приложения, учитывая соответствующий уровень персонализации и конфигурации для пользователя. Не диктуйте, а позволяйте пользователям определять стиль взаимодействия с приложением, но при этом не перегружайте их ненужными опциями и настройками, что может сбить с толку. Определитесь с ключевыми сценариями и сделайте их предельно простыми; обеспечьте простоту поиска информации и использования приложения.

2. Развитие информационных технологий.

Используйте преимущества развития ИТ-рынка, применяя существующие платформы и технологии. Создавайте приложения в настолько высокоуровневых средах, насколько это позволяют предъявляемые к приложениям требования, это позволит сконцентрироваться на том, что действительно уникально для вашего приложения, а не на воспроизведении уже существующих функций. Используйте шаблоны, являющиеся богатыми источниками проверенных решений распространенных проблем.

3. Гибкий дизайн.

Все большую популярность приобретают гибкие дизайны, использующие слабое связывание, что делает возможным повторное использование и упрощает поддержку. Архитектура с возможностью подключения модулей позволяет реализовать расширяемость после развертывания. Также для обеспечения взаимодействия с другими системами можно использовать преимущества таких сервис-ориентированных техник, как SOA.

4. Тенденции.

При построении архитектуры необходимо понимать тенденции, которые могут оказать влияние на дизайн после развертывания. Например, тенденции в насыщенных UI и мультимедиа, составных приложениях, увеличение пропускной способности и доступности сетей, все большее распространение мобильных устройств, продолжающийся рост производительности оборудования, интерес к моделям блогов и сообществ, рост популярности вычислений в облаке и удаленной работы.

На практике часто применяется подход, используемый в различных методологиях разработки ПО, и базирующийся на определении групп требований к продукту. Такой подход обычно включает группы (типы, категории) требований, например: системные, программные, функциональные, нефункциональные (в частности, атрибуты качества). Классический пример (см. рис. 1.2) высокоуровневого структурирования групп требований как требований к продукту описан в [16]. Требования к ПО состоят из трех уровней: бизнес-требования, требования пользователей и функциональные требования.

Также каждая система имеет свои нефункциональные требования. Модель на рис. 1.2 схематично иллюстрирует способ представления этих типов требований. Овалы обозначают типы информации для требований, а прямоугольники – способ хранения информации (документы, диаграммы, базы данных).

Рис. 1.2 Уровни требований

Бизнес-требования (business requirements) содержат высокоуровневые цели организации или заказчиков системы. Как правило, их высказывают те, кто финансируют проект, покупатели системы, менеджер реальных пользователей. В этом документе объясняется, почему организации нужна такая система, то есть описаны цели, которые организация намерена достичь с ее помощью. Рекомендуется [16] записывать бизнес-требования в форме документа об образе и границах проекта, который еще иногда называют уставом проекта (project charter) или документом рыночных требований (market requirements document). Определение границ проекта представляет собой первый этап управление общими проблемами расползания границ.

Требования пользователей (user requirements) описывают цели и задачи, которые пользователям позволит решить система. К отличным способам представления этого вида требований относятся варианты использования, сценарии и таблицы «событие-отклик», указывается, что клиенты смогут делать с помощью системы. Пример варианта использования - «Сделать заказ» для заказа билетов на самолет, аренды автомобиля.

Функциональные требования (functional requirements) определяет функциональность ПО, которую разработчики должны построить, чтобы пользователи смогли выполнить свои задачи в рамках бизнес-требований. Функциональные требования иногда называют требованиями поведения (behavioral requirements), они содержат положения с традиционным «должен» или «должна»: «Система должна по электронной почте отправлять пользователю подтверждение о заказе». Таким образом, функциональные требования описывают, что разработчику необходимо реализовать.

Термином системные требования (system requirements) обозначают высокоуровневые требования к продукту. Говоря о системе, подразумевается программное обеспечение или подсистемы ПО и оборудования. Люди тоже часть системы, поэтому определенные функции системы могут распространиться и на людей.

Из требований для всей системы можно вывести производные функциональные требования к ПО.

Бизнес-правила (business rules) включают корпоративные политики, правительственные постановления, промышленные стандарты и вычислительные алгоритмы. Бизнес-правила являются требованиями к ПО, потому что они находятся снаружи границ любой системы ПО. Однако они часто налагают ограничения, определяя, кто может выполнять конкретные варианты использования, или диктовать, какими функциями должна обладать система, подчиняющаяся соответствующим правилам. Иногда бизнес-правила становятся источником атрибутов качества, которые реализуются в функциональности. Следовательно, можно отследить происхождение конкретных функциональных требований вплоть до соответствующих им бизнес-правил.

Функциональные требования документируются в спецификации требований к ПО (software requirements specification, SRS), где описывается так полно, как необходимо, ожидаемое поведение системы. Формально это документ, хотя это может быть база данных или крупноформатная таблица с требованиями, хранилище данных в коммерческом инструменте управления требованиями или даже, может быть, набор карточек для небольшого проекта. Спецификация требований к ПО используется при разработке, тестировании, гарантии качества продукта, управлении проектом и связанных с проектом функциях.

В дополнение к функциональным требованиям спецификация содержит нефункциональные, где описаны цели и атрибуты качества. Атрибуты качества (quality attributes) представляют собой дополнительное описание функций продукта, выраженное через описание его характеристик, важных для пользователей или разработчиков. К таким характеристикам относятся легкость и простота использования, легкость перемещения, целостность, эффективность и устойчивость к сбоям. Другие нефункциональные требования описывают внешние взаимодействия между системой и внешним миром, а также ограничения дизайна и реализации. Ограничения (constraints) касаются выбора возможности разработки внешнего вида и структуры продукта.

Характеристика (feature) продукта – это набор логически связанных функциональных требований, которые обеспечивают возможности пользователя и удовлетворяют бизнес-цели. В области коммерческого ПО характеристика представляет собой узнаваемую всеми заинтересованными лицами группу требований, которые важны при принятии решения о покупке – элемент маркированного списка в описании продукта. Характеристики продукта, которые перечисляет клиент, не эквивалентны тем, что входят в список необходимых для решения задач пользователей. В качестве примеров характеристик продуктов можно привести избранные страницы или закладки Web-браузера, контроль за орфографией, запись макрокоманды, сервопривод стекла в автомобиле, онлайновое обновление или изменение налогового кодекса, ускоренный набор телефонного номера или автоматическое определение вируса. Характеристики могут охватывать множество вариантов использования, и для каждого варианта необходимо, чтобы множество функциональных требований было реализовано для выполнения пользователем его задач.

Рассмотрим пример [16]. Программа создания текстов.

Бизнес-требование может выглядеть так: «Продукт позволит пользователям исправлять орфографические ошибки в тексте эффективно». На коробке указано, что проверка грамматики включена как характеристика, удовлетворяющая бизнес-требования. Соответствующие требования пользователей могут содержать задачи (варианты использования): «Найдите орфографическую ошибку» или «Добавьте слово в общий словарь». Проверка грамматики имеет множество индивидуальных функциональных требований, которые связаны с такими операциями, как поиск и выделение слова с ошибкой, отображение диалогового окна с фрагментом текста, где это слово находится, и замена слова с ошибкой корректным вариантом по всему тексту. Атрибут качества легкость и простота использования (usability) как раз и определяет его значение посредством слова «эффективно» в бизнес-требованиях.

Таким образом, менеджеры и сотрудники отдела маркетинга определяют бизнес-требования для ПО, которые помогут их компании работать эффективнее (для информационных систем) или успешно конкурировать на рынке (для коммерческих продуктов). Каждое требование пользователя должно быть сопоставлено бизнес-требованию. На основе требований пользователя аналитики определяют функции, которые позволят пользователям выполнять их задачи. Разработчикам необходимы функциональные и нефункциональные требования, чтобы создавать решения с желаемой функциональностью, определенным качеством и требуемыми рабочими характеристиками, не выходя за рамки налагаемых ограничений.

Описанный процесс взаимодействия бизнес-требований, требований пользователей и функциональными требованиями на практике, как правило, итерационен и цикличен.

Отличия архитектуры от дизайна

Исходя из многочисленных определений, можно сделать вывод о том, что характерной чертой архитектуры является то, что ее трудно изменить на поздних

стадиях разработки. Связь между архитектурой и дизайном можно представить как пирамиду (рис. 1.3):

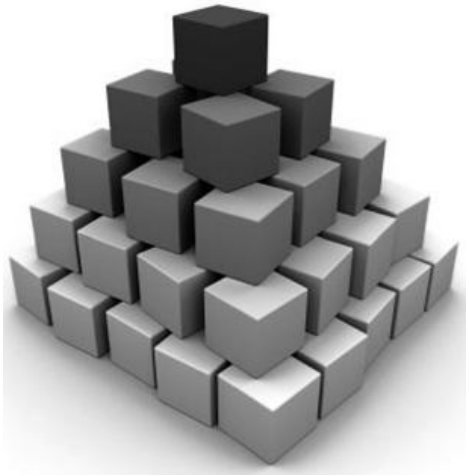

Рис. 1.3 Различия между архитектурой и дизайном

Архитектура приложения, показанная в виде светло серых кубиков на рисунке 1.3, представляет собой фундамент, на который опираются все остальные компоненты системы, в том числе элементы дизайна, показанные в виде темных кубиков. Архитектурные компоненты, как правило, трудно передвигать с места на место, поскольку при этом приходится двигать все лежащие выше блоки, которые будут затронуты изменениями в фундаменте.

В этом и заключается основное отличие архитектуры от дизайна. Например, Web-инфраструктура, использующаяся при создании Web-приложения, является элементом архитектуры, поскольку ее сложно сменить. При этом вы можете применять разные подходы к проектированию дизайна в рамках одной архитектуры, позволяющие достичь разные цели. Это говорит о том, что большинство приемов проектирования (design patterns) относится именно к дизайну, а не к архитектуре.

1.2 Основные принципы проектирования архитектуры ПО[2]

Рекомендации при проектировании архитектуры

В настоящее время при продумывании архитектуры предполагается, что дизайн будет эволюционировать со временем и что совершенно невозможно наперед знать все то, что необходимо для проектирования системы. Как правило, дизайн изменяется и дорабатывается в ходе реализации приложения по мере выявления новых сведений и в ходе тестирования на соответствие требованиям реального окружения. Создавайте архитектуру, ориентируясь на такие изменения, чтобы иметь возможность адаптировать их к требованиям, которые в начале процесса проектирования известны не в полном объеме.

При проектировании архитектуры необходимо ответить на следующие вопросы:

1. Какие части архитектуры являются фундаментальными, изменение которых в случае неверной реализации представляет наибольшие риски?

[2] Семинар №2.

2. Какие части архитектуры вероятнее всего подвергнуться изменениям, а также проектирование каких частей можно отложить?
3. Основные допущения, и как они будут проверяться?
4. Какие условия могут привести к реструктуризации дизайна?

При проектировании архитектуры руководствуйтесь следующими основными рекомендациями:

1. Создавайте, чтобы изменять.

Продумайте, как со временем может понадобиться изменить приложение, чтобы оно отвечало вновь возникающим требованиям и задачам, и предусмотрите необходимую гибкость.

2. Создавайте модели для анализа и сокращения рисков.

Используйте средства проектирования, системы моделирования, такие как Унифицированный язык моделирования (Unified Modeling Language, UML), и средства визуализации, когда необходимо выявить требования, принять архитектурные и проектные решения и проанализировать их последствия. Тем не менее, не создавайте слишком формализованную модель, она может ограничить возможности для выполнения итераций и адаптации дизайна.

3. Используйте модели и визуализации как средства общения при совместной работе.

Для построения хорошей архитектуры критически важен эффективный обмен информацией о дизайне, принимаемых решениях и вносимых изменениях. Используйте модели, представления и другие способы визуализации архитектуры для эффективного обмена информацией и связи со всеми заинтересованными сторонами, а также для обеспечения быстрого оповещение об изменениях в дизайне.

4. Выявляйте ключевые инженерные решения.

Используйте информацию, чтобы понять ключевые инженерные решения и области, в которых чаще всего возникают ошибки. В самом начале проекта уделите достаточное количество времени и внимания для принятия правильных решений, это обеспечит создание более гибкого дизайна, внесение изменений в который не потребует полной его переработки.

5. Не пытайтесь создать слишком сложную архитектуру и не делайте предположений, которые не можете проверить.

Лучше оставляйте свои варианты открытыми для изменения в будущем. Некоторые аспекты дизайна должны быть приведены в порядок на ранних стадиях процесса, потому что их возможная переработка может потребовать существенных затрат. Такие области необходимо выявить как можно раньше и уделить им достаточное количество времени.

6. Рассмотрите возможность использования инкрементного и итеративного подхода при работе над архитектурой.

Начинайте с базовой архитектуры, правильно воссоздавая полную картину, и затем прорабатывайте возможные варианты в ходе итеративного тестирования и доработки архитектуры. Не пытайтесь сделать все сразу; проектируйте настолько, насколько это необходимо для начала тестирования вашего дизайна на соответствие требованиям и допущениям. Усложняйте

дизайн постепенно, в процессе многократных пересмотров, чтобы убедиться, прежде всего, в правильности принятых крупных решений и лишь затем сосредотачиваться на деталях. Общей ошибкой является быстрый переход к деталям при ошибочном представлении о правильности крупных решений из-за неверных допущений или неспособности эффективно оценить свою архитектуру.

7. При тестировании архитектуры дайте ответы на следующие вопросы:
   A. Какие допущения были сделаны в этой архитектуре?
   B. Каким явным или подразумеваемым требованиям отвечает данная архитектура?
   C. Основные риски при использовании такого архитектурного решения?
   D. Каковы меры противодействия для снижения основных рисков?
   E. Является ли данная архитектура улучшением базовой архитектуры или одним из возможных вариантов архитектуры?

Основные принципы проектирования

Применение основных принципов проектирования при работе над архитектурой приложения поможет создать архитектуру, которая обеспечит минимизацию затрат, простоту обслуживания, удобство использования и расширяемость [6].

Рассмотрим основные принципы:

1. Разделение функций.

Разделите приложение на отдельные компоненты по возможности, минимальным перекрытием функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (high cohesion) и слабую связанность (low coupling). Неверное разграничение функциональности может привести к высокой связанности и сложностям взаимодействия, даже, несмотря на слабое перекрытие функциональности отдельных компонентов.

Связность – термин, часто используемый для оценки логического единства функций класса или модуля.

Модуль или класс обладает высокой связностью, если он спроектирован для выполнения группы взаимосвязанных функций. Классы с низкой связностью проектируются на основе набора разрозненных функций.

Концепция связности является более общей, чем принцип одной обязанности (единственности ответственности), но эти два понятия тесно связаны. Классы, соответствующие принципу, обычно обладают высокой связностью, и более просты в сопровождении, чем классы со многими обязанностями и низкой связностью.

2. Принцип единственности ответственности.

Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.

3. Принцип минимального знания (также известный как Закон Деметера (Law of Demeter, LoD)).

Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.

4. Не повторяйтесь (Don't repeat yourself, DRY).

Намерение должно быть обозначено только один раз. В применении к проектированию приложения это означает, что определенная функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.

5. Минимизируйте проектирование наперед.

Проектируйте только то, что необходимо. В некоторых случаях, когда стоимость разработки или издержки в случае неудачного дизайна очень высоки, может потребоваться полное предварительное проектирование и тестирование. В других случаях, особенно при гибкой разработке, можно избежать масштабного проектирования наперед (big design upfront, BDUF). Если требования к приложению четко не определены, или существует вероятность изменения дизайна со временем, старайтесь не тратить много сил на проектирование раньше времени. Этот принцип называют YAGNI («You ain't gonna need it» - Вам это никогда не понадобится).

Таким образом, исходя из перечисленных принципов цель архитектора ПО при проектировании системы состоит в максимальном упрощении дизайна через его разбиение на функциональные области.

Например, к разным функциональным областям следует отнести пользовательский интерфейс (user interface, UI), выполнение бизнес-процессов и доступ к данным.

Компоненты в каждой из этих областей должны реализовывать данную конкретную функциональность и не должны смешивать в себе код разных функциональных областей. Так, например, в компонентах UI не должно быть кода прямого доступа к источнику данных, для извлечения данных в них должны использоваться либо бизнес-компоненты, либо компоненты доступа к данным.

Также необходимо проанализировать соотношение затрат/выгод для инвестиций в приложение. В некоторых случаях может быть целесообразным упростить структуру и разрешить, например, связывание элементов UI с результирующими данными. В общем, оценивайте реализацию функциональности также и с коммерческой точки зрения.

Далее приводятся обобщенные рекомендации (практики проектирования, рекомендации по построению слоев приложения, а также разработке компонентов и модулей), которые помогут учесть широкий диапазон факторов, влияющих на проектирование, реализацию, развертывание, тестирование и обслуживание приложения.

Практики проектирования

1. Придерживайтесь единообразия шаблонов проектирования в рамках одного слоя.

По возможности, в рамках одного логического уровня структура компонентов, выполняющих определенную операцию, должна быть единообразной. Например, если для объекта, выступающего в роли шлюза к таблицам или представлениям базы данных, решено использовать шаблон Table Data Gateway (Шлюз таблицы данных), не надо включать еще один шаблон, скажем, Repository (Хранилище), использующий другую парадигму доступа к данным и инициализации бизнес- сущностей. Однако для задач с более широким диапазоном требований может потребоваться применить разные шаблоны, например, для приложения, включающего поддержку бизнес-транзакций и составления отчетов.

2. Не дублируйте функциональность в приложении.

Та или иная функциональность должна обеспечиваться только одним компонентом, ее дублирование в любом другом месте недопустимо. Это обеспечивает связность компонентов и упрощает их оптимизацию в случае необходимости изменения отдельной функциональной возможности. Дублирование функциональности в приложении может усложнить реализацию изменений, снизить понятность приложения и создать потенциальную возможность для несогласованностей.

3. Предпочитайте композицию наследованию.

По возможности, для повторного использования функциональности применяйте композицию, а не наследование, потому что наследование увеличивает зависимость между родительским и дочерними классами, ограничивая, таким образом, возможности повторного использования последних. Это также будет способствовать уменьшению глубины иерархий наследования, что упростит работу с ними.

4. Применяйте определенный стиль написания кода и соглашение о присваивании имен для разработки.

Поинтересуйтесь, имеет ли организация сформулированный стиль написания кода и соглашения о присваивании имен. Если нет, необходимо придерживаться общепринятых стандартов. В этом случае вы получите единообразную модель, все участники группы смогут без труда работать с кодом, написанным не ими, т.е. код станет более простым и удобным в обслуживании.

5. Обеспечивайте качество системы во время разработки с помощью методик автоматизированного контроля качества (Quality Assurance - QA).

Используйте в процессе разработки модульное тестирование и другие методики автоматизированного Анализа качества (Quality Analysis), такие как анализ зависимостей и статический анализ кода. Четко определяйте показатели поведения и производительности для компонентов и подсистем и используйте автоматизированные инструменты QA в процессе разработки, чтобы гарантировать отсутствие неблагоприятного воздействия локальных решений по проектированию или реализации на качество всей системы.

6. Учитывайте условия эксплуатации приложения.

Определите необходимые ИТ-инфраструктуре показатели и эксплуатационные данные, чтобы гарантировать эффективное развертывание и работу приложения. Приступайте к проектированию компонентов и подсистем приложений, только имея ясное представление об их индивидуальных эксплуатационных требованиях, что существенно упростит общее развертывание и эксплуатацию. Использование автоматизированных инструментов QA при разработке гарантированно обеспечит получение необходимых эксплуатационных характеристик компонентов и подсистем приложения.

Слои приложения

1. Разделяйте функциональные области.

Разделите приложение на отдельные функции, по возможности, с минимальным перекрытием функциональности. Основное преимущество такого подхода – независимая оптимизация функциональных возможностей. Кроме того, сбой одной из функций не приведет к сбою остальных, поскольку они могут выполняться независимо друг от друга. Такой подход также упрощает понимание и проектирование приложения и облегчает управление сложными взаимосвязанными системами.

2. Явно определяйте связи между слоями.

Решение, в котором каждый слой приложения может взаимодействовать или имеет зависимости со всеми остальными слоями, является сложным для понимания и управления. Принимайте явные решения о зависимостях между слоями и о потоках данных между ними.

2. Реализуйте слабое связывание слоев с помощью абстракции.

Это можно реализовать, определяя интерфейсные компоненты с хорошо известными входными и выходными характеристиками, такие как фасад, которые преобразуют запросы в формат, понятный компонентам слоя. Кроме того, также можно определять общий интерфейс или совместно используемую абстракцию (противоположность зависимости), которые должны быть реализованы компонентами интерфейса, используя интерфейсы или абстрактные базовые классы.

3. Не смешивайте разные типы компонентов на одном логическом уровне.

Начинайте с идентификации функциональных областей и затем группируйте компоненты, ассоциированные с каждой из этих областей в логические уровни. Например, слой UI не должен включать компоненты выполнения бизнес-процессов, в него должны входить только компоненты, используемые для обработки пользовательского ввода и запросов.

4. Придерживайтесь единого формата данных в рамках слоя или компонента.

Смешение форматов данных усложнит реализацию, расширение и обслуживание приложения. Любое преобразование одного формата данных в другой требует реализации кода преобразования и влечет за собой издержки на обработку.

Рекомендации для компонентов, модулей и функций

1. Компонент или объект не должен полагаться на внутренние данные других компонентов или объектов.

Каждый метод, вызываемый компонентом или методом другого объекта или компонента, должен располагать достаточными сведениями о том, как обрабатывать поступающие запросы и, в случае необходимости, как перенаправлять их к соответствующим подкомпонентам или другим компонентам. Это способствует созданию более удобных в обслуживании и адаптируемых приложений.

2. Не перегружайте компонент функциональностью.

Например, компонент UI не должен включать код для доступа к данным или обеспечивать дополнительную функциональность. Перегруженные компоненты часто имеют множество функций и свойств, сочетая бизнес-функциональность и сквозную функциональность, такие как протоколирование и обработка исключений. В результате получается очень неустойчивый к ошибкам и сложный в обслуживании дизайн. Применение принципов исключительной ответственности и разделения функциональности поможет избежать этого.

3. Разберитесь с тем, как будет осуществляться связь между компонентами.

Это требует понимания сценариев развертывания, которые должно поддерживать создаваемое приложение. Необходимо определить, будут ли все компоненты выполняться в рамках одного процесса или необходимо обеспечить поддержку связи через физические границы или границы процесса, вероятно, путем реализации интерфейсов взаимодействия на основе сообщений.

4. Максимально изолируйте сквозную функциональность от бизнес-логики приложения.

Сквозная функциональность - это аспекты безопасности, обмена информацией или управляемости, такие как протоколирование и инструментирование. Смешение кода, реализующего эти функции, с бизнес-логикой может привести к созданию дизайна, который будет сложно расширять и обслуживать. Внесение изменений в сквозную функциональность потребует переработки всего кода бизнес-лоигики. Рассмотрите возможность использования инфраструктур и методик (таких как аспект-ориентированное программирование), которые помогут в реализации такой функциональности.

5. Определяйте четкий контракт для компонентов.

Компоненты, модули и функции должны определять контракт или спецификацию интерфейса, четко оговаривающую их использование и поведение. Контракт должен описывать, как другие компоненты могут выполнять доступ к внутренней функциональности компонента, модуля или функции, и поведение этой функциональности с точки зрения предварительных условий, постусловий, побочных эффектов, исключений, рабочих характеристик и других факторов.

Основные принимаемые решения при реализации процесса проектирования[3]

Основными решениями, которые должны быть приняты, и которые помогут учесть все важные факторы при реализации итеративной разработки проекта архитектуры, являются следующие:

1. Определение типа приложения.
2. Выбор стратегии развертывания.
3. Выбор соответствующих технологий.
4. Выбор показателей качества.
5. Решение о путях реализации сквозной функциональности.

Определение типа приложения

Выбор соответствующего типа приложения это ключевой момент процесса проектирования приложения. Данный выбор определяется конкретными требованиями и ограничениями среды. От многих приложений требуется поддержка множества типов клиентов и возможность использования более одного базового архетипа. В основном используются следующие основные типы приложений:

- приложения для мобильных устройств;
- насыщенные клиентские приложения для выполнения преимущественно на клиентских ПК;
- насыщенные клиентские приложения для развертывания из интернета с поддержкой насыщенных UI и мультимедийных сценариев;
- сервисы, разработанные для обеспечения связи между слабо связанными компонентами;
- веб-приложения для выполнения преимущественно на сервере в сценариях с постоянным подключением.

Кроме того, в руководстве [6] представлены сведения и рекомендации по некоторым более специальным типам приложений:

- приложения и сервисы, размещаемые в центрах обработки данных (ЦОД) и в облаке;
- офисные бизнес-приложения (Office Business Applications, OBAs), интегрирующие технологии Microsoft Office и Microsoft Server;
- бизнес-приложения SharePoint (SharePoint Line of Business, LOB), обеспечивающие доступ к бизнес-данным и функциональным возможностям через портал.

Выбор стратегии развертывания

Приложение может развертываться в разнообразнейших средах, каждая из которых будет иметь собственный набор ограничений, таких как физическое распределение компонентов по серверам, ограничение по используемым сетевым протоколам, настройки межсетевых экранов и маршрутизаторов и многое другое. Существует несколько общих схем развертывания, которые описывают преимущества и мотивы применения ряда распределенных и нераспределенных сценариев. При выборе стратегии необходимо найти

[3] Домашнее задание

компромисс между требованиями приложения и соответствующими схемами развертывания, поддерживаемым оборудованием, и ограничениями, налагаемыми средой на варианты развертывания. Все эти факторы будут влиять на проектируемую архитектуру.

Выбор соответствующих технологий

Ключевым фактором при выборе технологий для приложения является тип разрабатываемого приложения, а также предпочтительные варианты топологии развертывания приложения и архитектурные стили. Выбор технологий также определяется политиками организации, ограничениями среды, квалификацией ресурсов и т.д. Необходимо сравнить возможности выбираемых технологий с требованиями приложения, принимая во внимание все эти факторы.

Выбор показателей качества

Показатели качества, такие как безопасность, производительность, удобство и простота использования, помогают сфокусировать внимание на критически важных проблемах, которые должен решать создаваемый дизайн.

В зависимости от конкретных требований может понадобиться рассмотреть все показатели качества или только некоторые из них. Например, вопросы безопасности и производительности необходимо учесть при разработке каждого приложения, тогда как проблемы возможности взаимодействия или масштабируемости стоят далеко не перед всеми проектами. В первую очередь, необходимо понять поставленные требования и сценарии развертывания, чтобы знать, какие показатели качества важны для создаваемого приложения. Нельзя также забывать о возможности противоречий между показателями качества. Например, часто требования безопасности идут вразрез с производительностью или удобством использования.

При проектировании с учетом показателей качества следует руководствоваться следующими соображениями:

- показатели качества – это свойства системы, отделенные от ее функциональности;
- с технической точки зрения показатели качества отличают хорошую систему и этим помогают построить правильную архитектуру;
- существует два типа показателей качества: измеряемые во время выполнения и не измеряемые, т.е. те, оценить которые можно только посредством испытаний разработанной системы;
- необходимо провести анализ и найти оптимальное соотношение между показателями качества.

Сделать выбор показателей качества могут помощь ответы на следующие вопросы:

1. Каковы основные показатели качества приложения?
2. Каковы основные требования для реализации этих показателей? Поддаются ли они количественному определению?
3. Каковы критерии приемки, которые будут свидетельствовать о выполнении требований?

Решение о путях реализации сквозной функциональности

Сквозная функциональность представляет ключевую область дизайна, не связанную с конкретным функционалом приложения. Например, необходимо рассмотреть возможности реализации централизованных или общих решений для следующих аспектов:

- механизм протоколирования, обеспечивающий возможность каждому слою вести журнал в общем хранилище либо в разных хранилищах, но таким образом, чтобы результаты могли быть сопоставлены впоследствии;
- механизмы аутентификации и авторизации, обеспечивающие передачу удостоверений на разные уровни для предоставления доступа к ресурсам;
- инфраструктура управления исключениями, которая будет функционировать в каждом слое и между уровнями, если исключения распространяются в рамках системы;
- подход к реализации связей, используемый для обеспечения обмена информацией между слоями;
- общая инфраструктура кэширования, позволяющая кэшировать данные в слое представления, бизнес-слое и слое доступа к данным.

1.3 Архитектурные шаблоны и стили

В данном вопросе описываются обобщенные шаблоны и принципы, применяемые при проектировании современных приложений. Часто их называют архитектурными стилями или парадигмами. К ним относят такие шаблоны как клиент/сервер, многоуровневая архитектура, компонентная архитектура, архитектура, основанная на шине сообщений, и сервисно-ориентированная архитектура (service-oriented architecture, SOA).

Для каждого стиля предлагается обзор, основные принципы, основные преимущества и сведения, которые помогут выбрать подходящие для приложения архитектурные стили. Важно понимать, что стили описывают разные аспекты приложений. Некоторые архитектурные стили описывают схемы развертывания, некоторые вопросы структуры и дизайна, а другие аспекты связи. Таким образом, типовое приложение, как правило, использует сочетание нескольких рассматриваемых ниже стилей.

Архитектурный стиль, иногда называемый архитектурным шаблоном (высокоуровневая схема) – это набор принципов, обеспечивающая абстрактную инфраструктуру для семейства систем.

Архитектурный стиль определяет набор компонентов и соединений, которые могут использоваться в экземплярах этого стиля, а также ряд ограничений по их возможным сочетаниям. Сюда могут относиться топологические ограничения на архитектурные решения (например, не использовать циклы). Описание стиля также может включать и другие ограничения, такие как, необходимость обработки семантики выполнения.

Архитектурный стиль улучшает секционирование и способствует повторному использованию дизайна благодаря обеспечению решений часто

встречающихся проблем. Архитектурные стили и шаблоны можно рассматривать как набор принципов, формирующих приложение.

Понимание архитектурных стилей обеспечивает несколько преимуществ. Самое главное из них – общий язык. Также они дают возможность вести диалог, не касаясь технологий, т.е. обсуждать схемы и принципы, не вдаваясь в детали. Например, архитектурные стили позволяют сравнивать схему клиент/сервер с n-уровневой схемой приложения.

Архитектурные стили можно организовать по категориям. В таблице 1.1 перечислены основные категории и соответствующие архитектурные стили.

Таблица 1.1

Категория	**Архитектурные стили**
Связь	Сервисно-ориентированная архитектура (SOA), шина сообщений
Развертывание	Клиент/сервер, N-уровневая, 3-уровневая
Предметная область	Дизайн на основе предметной области (Domain Driven Design)
Структура	Компонентная, объектно-ориентированная, многоуровневая

В таблице 1.2 приводится список типовых архитектурных стилей, и дается краткое описание каждого из них. Далее эти стили раскрываются подробно, а также даются рекомендации по выбору соответствующих стилей для конкретного приложения.

Таблица 1.2

Архитектурный стиль/парадигма	**Описание**
Клиент/сервер	Система разделяется на два приложения, где клиент выполняет запросы к серверу. Во многих случаях в роли сервера выступает база данных, а логика приложения представлена
Компонентная архитектура	Дизайн приложения разлагается на функциональные или логические компоненты с возможностью повторного использования, предоставляющие тщательно проработанные интерфейсы связи.
Дизайн на основе предметной области	Объектно-ориентированный архитектурный стиль, ориентированный на моделирование сферы деловой активности и определяющий бизнес-объекты на основании сущностей этой сферы.
Многослойная архитектура	Функциональные области приложения разделяются на многослойные группы (уровни).

Архитектурный стиль/парадигма	**Описание**
Шина сообщений	Архитектурный стиль, предписывающий использование программной системы, которая может принимать и отправлять сообщения по одному или более каналам связи, так что приложения получают возможность взаимодействовать, не располагая конкретными сведениями друг о друге.
N-уровневая / 3-уровневая	Функциональность выделяется в отдельные сегменты, во многом аналогично многослойному стилю, но в данном случае сегменты физически располагаются на разных
Объектно-ориентированная	Парадигма проектирования, основанная на распределении ответственности приложения или системы между отдельными многократно используемыми и самостоятельными объектами, содержащими данные и поведение.
Сервисно-ориентированная архитектура (SOA)	Описывает приложения, предоставляющие и потребляющие функциональность в виде сервисов с помощью контрактов и сообщений.

Архитектура программной системы практически никогда не ограничена лишь одним архитектурным стилем. Часто она является сочетанием архитектурных стилей, образующих полную систему. Например, может существовать SOA-дизайн, состоящий из сервисов, при разработке которых использовалась многослойная архитектура и объектно-ориентированный архитектурный стиль.

Сочетание архитектурных стилей также полезно при построении интернет-приложений, где можно достичь эффективного разделения функциональности за счет применения многослойного архитектурного стиля. Таким образом можно отделить логику представления от бизнес-логики и логики доступа к данным. Требования безопасности организации могут обусловливать либо 3-уровневое развертывание приложения, либо развертывание с более чем тремя уровнями. Уровень представления может развертываться в пограничной сети, располагающейся между внутренней сетью организации и внешней сетью. В качестве модели взаимодействия на уровне представления может применяться шаблон представления с отделением (разновидность многослойного стиля), такая как Model-View-Controller (MVC). Также можно выбрать архитектурный стиль SOA и реализовать связь между Веб-сервером и сервером приложений посредством обмена сообщениями.

Создавая настольное приложение, можно реализовать клиент, который будет отправлять запросы к программе на сервере. В этом случае развертывание

клиента и сервера можно выполнить с помощью архитектурного стиля клиент/сервер и использовать компонентную архитектуру для дальнейшего разложения дизайна на независимые компоненты, предоставляющие соответствующие интерфейсы. Применение объектно-ориентированного подхода к этим компонентам повысит возможности повторного использования, тестирования и гибкость.

На выбор архитектурных стилей оказывает влияние множество факторов. Сюда входят способность организации к проектированию и реализации, возможности и опыт разработчиков, а также ограничения инфраструктуры и организации.

Архитектура клиент/сервер

Клиент/серверная архитектура описывает распределенные системы, состоящие из отдельных клиента и сервера и соединяющей их сети.

Простейшая форма системы клиент/сервер, называемая 2-уровневой архитектурой – это серверное приложение, к которому напрямую обращаются множество клиентов.

Исторически архитектура клиент/сервер представляет собой настольное приложение с графическим UI, обменивающееся данными с сервером базы данных, на котором в форме хранимых процедур располагается основная часть бизнес-логики, или с выделенным файловым сервером. Если рассматривать более обобщенно, архитектурный стиль клиент/сервер описывает отношения между клиентом и одним или более серверами, где клиент инициирует один или более запросов (возможно, с использованием графического UI), ожидает ответы и обрабатывает их при получении. Обычно сервер авторизует пользователя и затем проводит обработку, необходимую для получения результата. Для связи с клиентом сервер может использовать широкий диапазон протоколов и форматов данных.

На сегодняшний день примерами архитектурного стиля клиент/сервер могут служить:

- веб-приложения, выполняющиеся в Интернете или внутренних сетях организаций;
- настольные приложения для операционной системы Windows, выполняющие доступ к сетевым сервисам данных;
- приложения, выполняющие доступ к удаленным хранилищам данных (такие как программы чтения электронной почты, FTP-клиенты и средства доступа к базам данных);
- инструменты и утилиты для работы с удаленными системами (такие как средства управления системой и средства мониторинга сети).

Также могут использоваться и другие разновидности стиля клиент/сервер:

1. Системы клиент-очередь-клиент.

Этот подход позволяет клиентам обмениваться данными с другими клиентами через очередь на сервере. Клиенты могут читать данные с и отправлять данные на сервер, который выступает в роли простой очереди для хранения данных. Благодаря этому клиенты могут распределять и

синхронизировать файлы и сведения. Иногда такую архитектуру называют пассивной очередью.

2. Одноранговые (Peer-to-Peer, P2P) приложения.

Созданный на базе клиент-очередь- клиент, стиль P2P позволяет клиенту и серверу обмениваться ролями с целью распределения и синхронизации файлов и данных между множеством клиентов. Эта схема расширяет стиль клиент/сервер, добавляя множественные ответы на запросы, совместно используемые данные, обнаружение ресурсов и устойчивость при удалении участников сети.

3. Серверы приложений.

Специализированный архитектурный стиль, при котором приложения и сервисы размещаются и выполняются на сервере, и тонкий клиент выполняет доступ к ним через браузер или специальное установленное на клиенте ПО. Примером является клиент, который работает с приложением, выполняющимся на сервере, через такую среду как Terminal Services (Службы терминалов).

Основные преимущества архитектурного стиля клиент/сервер

1. Достаточно высокий уровень безопасности.

Все данные хранятся на сервере, который обычно обеспечивает больший контроль безопасности, чем клиентские компьютеры.

2. Централизованный доступ к данным.

Поскольку данные хранятся только на сервере, администрирование доступа к данным намного проще, чем в любых других архитектурных стилях.

3. Простота обслуживания.

Роли и ответственность вычислительной системы распределены между несколькими серверами, общающимися друг с другом по сети. Благодаря этому клиент гарантированно остается неосведомленным и не подверженным влиянию событий, происходящих с сервером (ремонт, обновление либо перемещение).

Рекомендации к применению

Рассмотрите возможность применения архитектуры клиент/сервер в следующих случаях:

- создаваемое приложение должно размещаться на сервере и не должно поддерживать множество клиентов;
- создаются Веб-приложения, предоставляемые через Веб-браузер;
- реализуются бизнес-процессы, которые будут использоваться в рамках организации;
- создаются сервисы для использования другими приложениями.

Архитектурный стиль клиент/сервер, как и многие стили сетевых приложений, также подходит, если необходимо централизовать хранилище данных, функции резервного копирования и управления, или если разрабатываемое приложение должно поддерживать разные типы клиентов и разные устройства.

Тем не менее, традиционная 2-уровневая архитектура клиент/сервер имеет множество недостатков, включая тенденцию тесного связывания данных и бизнес-логики приложения на сервере, что может иметь негативное влияние на расширяемость и масштабируемость системы, и зависимость от центрального сервера, что негативно сказывается на надежности системы.

Для решения этих проблем архитектурный стиль клиент/сервер был развит в более универсальный 3-уровневый (или N-уровневый), описываемый ниже, в котором устранены некоторые недостатки, свойственные 2-уровневой архитектуре клиент/сервер, и обеспечиваются дополнительные преимущества.

Компонентная архитектура

Компонентная архитектура описывает подход к проектированию и разработке систем с использованием методов проектирования программного обеспечения.

Основное внимание в этом случае уделяется разложению дизайна на отдельные функциональные или логические компоненты, предоставляющие четко определенные интерфейсы, содержащие методы, события и свойства. В данном случае обеспечивается более высокий уровень абстракции, чем при объектно-ориентированной разработке, и не происходит концентрации внимания на таких вопросах, как протоколы связи или общее состояние.

Основной принцип компонентного стиля – применение компонентов, обладающих следующими качествами:

– Пригодность для повторного использования.

Как правило, компоненты проектируются с обеспечением возможности их повторного использования в разных сценариях различных приложений. Однако некоторые компоненты создаются специально для конкретной задачи.

– Замещаемость.

Компоненты могут без труда заменяться другими подобными компонентами.

– Независимость от контекста.

Компоненты проектируются для работы в разных средах и контекстах. Специальные сведения, такие как данные о состоянии, должны не включаться или извлекаться компонентом, а передаваться в него.

– Расширяемость.

Компонент может расширять существующие компоненты для обеспечения нового поведения.

– Инкапсуляция.

Компоненты предоставляют интерфейсы, позволяющие вызывающей стороне использовать их функциональность, не раскрывая при этом детали внутренних процессов либо внутренние переменные или состояние.

– Независимость.

Компоненты проектируются с минимальными зависимостями от других компонентов. Таким образом, компоненты могут быть развернуты в любой подходящей среде без влияния на другие компоненты или системы.

Обычно в приложениях используются компоненты пользовательского интерфейса (их часто называют элементами управления), такие как таблицы и кнопки, а также вспомогательные или служебные компоненты, предоставляющие определенный набор функций, используемых в других компонентах. К другому типу распространенных компонентов относятся ресурсоемкие компоненты, доступ к которым осуществляется нечасто, и

активация которых выполняется «точно вовремя» (just-in-time, JIT) (обычно используется в сценариях с удаленными или распределенными компонентами); и компоненты с очередью, вызовы методов которых могут выполняться асинхронно за счет применения очереди сообщений, для хранения и пересылки.

Компоненты зависят от механизмов платформы, обеспечивающей среду выполнения. Часто эти механизмы называют просто компонентной архитектурой. Примерами такой архитектуры могут служить Объектная модель программных компонентов (component object model, COM) и Объектная модель распределенных программных компонентов (distributed component object model, DCOM) в Windows, Общая архитектура брокера объектных запросов (Common Object Request Broker Architecture, CORBA) и Серверные компоненты Java (Enterprise JavaBeans, EJB) на других платформах. Используемая компонентная архитектура описывает механизмы размещения компонентов и их интерфейсов, передачи сообщений или команд между компонентами и, в некоторых случаях, сохранения состояния.

Однако термин компонент часто используется в более общем смысле как составная часть, элемент или составляющая. Microsoft .NET Framework предоставляет поддержку для построения приложений с использованием такого компонентного подхода. Например, бизнес-компоненты и компоненты данных, которые обычно являются классами, скомпилированными в сборки .NET Framework. Они выполняются под управлением среды выполнения .NET Framework. Каждая сборка может включать более одного подобного компонента.

Основные преимущества компонентного архитектурного стиля

– Простота развертывания.

Существующие версии компонентов могут заменяться новыми совместимыми версиями, не оказывая влияния на другие компоненты или систему в целом.

– Меньшая стоимость.

Использование компонентов сторонних производителей позволяет распределять затраты на разработку и обслуживание.

– Простота разработки.

Для обеспечения заданной функциональности компоненты реализуют широко известные интерфейсы, что позволяет вести разработку без влияния на другие части системы.

– Возможность повторного использования.

Применение многократно используемых компонентов означает возможность распределения затрат на разработку и обслуживание между несколькими приложениями или системами.

– Упрощение с технической точки зрения.

Компоненты упрощают систему через использование контейнера компонентов и его сервисов. В качестве примеров сервисов, предоставляемых контейнером, можно привести активацию компонентов, управление жизненным

циклом, организацию очереди вызовов методов, обработку событий и транзакции.

Для управления зависимостями между компонентами, поддержки слабого связывания и повторного использования могут использоваться шаблоны проектирования, такие как Dependency Injection (Внедрение зависимостей) или Service Locator (Локатор сервиса). Такие шаблоны часто применяются при создании составных приложений, сочетающих и повторно использующих компоненты во многих приложениях.

Рекомендации к применению

Рассмотрите возможность применения компонентой архитектуры в следующих случаях:

- уже располагаете подходящими компонентами или можете получить их от сторонних производителей;
- предполагается, что создаваемое приложение будет преимущественно выполнять процедурные функции, возможно, с небольшим количеством вводимых данных или вообще без таковых;
- хотите иметь возможность сочетать компоненты, написанные на разных языках программирования.

Также этот стиль подойдет для создания подключаемой или составной архитектуры, которая позволяет без труда заменять и обновлять отдельные компоненты.

Проектирование на основе предметной области

Проектирование на основе предметной области (Domain Driven Design, DDD) – это объектно-ориентированный подход к проектированию ПО, основанный на предметной области, ее элементах, поведении и отношениях между ними [2].

Целью является создание программных систем, являющихся реализацией лежащей в основе предметной области, путем определения модели предметной области, выраженной на языке специалистов в этой области.

Модель предметной области может рассматриваться как каркас, на основании которого будут реализовываться решения.

Для применения DDD необходимо четко понимать предметную область, которую предполагается моделировать, или иметь способности для овладения такими знаниями. При создании модели предметной области группа разработки нередко работает в сотрудничестве со специалистами в данной области. Архитекторы, разработчики и специалисты в рассматриваемой области обладают разной подготовкой и во многих ситуациях будут использовать разные языки для описания своих целей, желаний и требований. Однако в рамках DDD вся группа договаривается использовать только один язык, ориентированный на предметную область и исключающий все технические жаргонизмы.

В качестве ядра ПО выступает модель предметной области, которая является прямой проекцией этого общего языка, с ее помощью путем анализа языка группа быстро находит пробелы в ПО. Создание общего языка это не просто упражнение по получению сведений от специалистов и их применению.

Довольно часто в группах возникают проблемы с обменом информацией не только по причине непонимания языка предметной области, но также и из-за неопределенности языка самого по себе. Процесс DDD имеет целью не только реализацию используемого языка, но также улучшение и уточнение языка предметной области. Это, в свою очередь, положительно отражается на создаваемом ПО, поскольку модель является прямой проекцией языка предметной области.

Чтобы сделать модель строгой и полезной языковой конструкцией, как правило, приходится интенсивно использовать изоляцию и инкапсуляцию в рамках модели предметной области.

Это может обусловить относительную дороговизну системы, основанной на DDD. Несмотря на то, что DDD обеспечивает массу преимуществ с технической точки зрения, таких как удобство в обслуживании, эта схема должна применяться лишь для сложных предметных областей, для которых процессы моделирования и лингвистического анализа обеспечивают безусловные преимущества при обмене сложной для понимания информацией и формулировании общего видения предметной области.

Основные преимущества стиля DDD являются

– Обмен информацией.

Все участники группы разработки могут использовать модель предметной области и описываемые ею сущности для передачи сведений и требований предметной области с помощью общего языка предметной области, не прибегая к техническому жаргону.

– Расширяемость.

Модель предметной области часто является модульной и гибкой, что упрощает обновление и расширение при изменении условий и требований.

– Удобство тестирования.

Объекты модели предметной области характеризуются слабой связанностью и высокой связность, что облегчает их тестирование.

Рекомендации к применению

Рассмотрите возможность применения DDD в следующих случаях:

– хотите улучшить процессы обмена информацией и ее понимания группой разработки;
– необходимо представить проект приложения на общем языке, понятном всем заинтересованным сторонам.

DDD также может быть идеальным подходом для сценариев обработки больших объемов сложных данных предприятия, с которыми сложно справится, используя другие методики.

Многослойная архитектура

Многоуровневая архитектура обеспечивает группировку связанной функциональности приложения в разных слоях, выстраиваемых вертикально, поверх друг друга.

Существует два похожих понятия: слой и уровень. Слой обозначает логическое разделение функциональности, а уровень физическое разворачивание на разных системах.

Функциональность каждого слоя объединена общей ролью или ответственностью. Слои слабо связаны, и между ними осуществляется явный обмен данными. Правильное разделение приложения на слои помогает поддерживать строгое разделение функциональности, что в свою очередь, обеспечивает гибкость, а также удобство и простоту обслуживания.

Многослойная архитектура описана как перевернутая пирамида повторного использования, в которой каждый слой агрегирует ответственности и абстракции уровня, расположенного непосредственно под ним. При строгом разделении на слои компоненты одного слоя могут взаимодействовать только с компонентами того же слоя или компонентами слоя, расположенного прямо под данным слоем. Более свободное разделение на слои позволяет компонентам слои взаимодействовать с компонентами того же и всех нижележащих слоев.

Слои приложения могут размещаться физически на одном компьютере (на одном уровне) или быть распределены по разным компьютерам (n-уровней), и связь между компонентами разных уровней осуществляется через строго определенные интерфейсы. Например, типовое Веб-приложение состоит из слоя представления (функциональность, связанная с UI), бизнес-слоя (обработка бизнес-правил) и слоя данных (функциональность, связанная с доступом к данным, часто практически полностью реализуемая с помощью высокоуровневых инфраструктур доступа к данным).

Рассмотрим общие принципы проектирования с использованием многослойной архитектуры:

– Абстракция.

Многослойная архитектура представляет систему как единое целое, обеспечивая при этом достаточно деталей для понимания ролей и ответственностей отдельных слоев и отношений между ними.

– Инкапсуляция.

Во время проектирования нет необходимости делать какие-либо предположения о типах данных, методах и свойствах или реализации, поскольку все эти детали скрыты в рамках слоя.

– Четко определенные функциональные слои.

Разделение функциональности между слоями очень четкая. Верхние слои, такие как слой представления, посылают команды нижним слоям, таким как бизнес-слой и слой данных, и могут реагировать на события, возникающие в этих слоях, обеспечивая возможность передачи данных между слоями вверх и вниз.

– Высокая связность.

Четко определенные границы ответственности для каждого слоя и гарантированное включение в слой только функциональности, напрямую связанной с его задачами, поможет обеспечить максимальную связность в рамках слоя.

– Возможность повторного использования.

Отсутствие зависимостей между нижними и верхними слоями обеспечивает потенциальную возможность их повторного использования в других сценариях.

– Слабое связывание. Для обеспечения слабого связывания между слоями связь между ними основывается на абстракции и событиях.

Примерами многослойных приложений могут служить бизнес-приложения (line-of-business, LOB), такие как системы бухгалтерского учета и управления заказчиками; Веб-приложения и Веб-сайты предприятий; настольные или смарт-клиенты предприятий с централизованными серверами приложений для размещения бизнес-логики.

Многослойная архитектура поддерживается рядом шаблонов проектирования [2,3]. Например, под названием Separated Presentation объединяется ряд шаблонов, разделяющих взаимодействие пользователя с UI, представление, бизнес-логику и данные приложения, с которыми работает пользователь. Отделение представления позволяет создавать UI в графических дизайнерах, в то время как разработчики пишут управляющий код. Такое разделение функциональности на роли повышает возможность тестирования поведения отдельных ролей.

Основными принципами шаблонов Separated Presentation являются:

– Разделение функциональности.

Шаблоны Separated Presentation разделяют функциональность UI на роли, например, в шаблоне MVC имеется три роли: Модель (Model), Представление (View) и Контроллер (Controller). Модель представляет данные (возможно, модель предметной области, которая включает базнес-правила). Представление отвечает за внешний вид UI. Контроллер обрабатывает запросы, работает с моделью и выполняет другие операции.

– Уведомление на основе событий. Шаблон Observer (Наблюдатель) обычно используется для уведомления Представления об изменениях данных, управляемых Моделью.
– Делегированная обработка событий. Контроллер обрабатывает события, формируемые элементами управления UI в Представлении.

В качестве примеров шаблонов Separated Presentation можно отметить шаблон Passive View (Пассивное представление) и шаблон Supervising Presenter (Наблюдающий презентатор), также называемый Supervising Controller (Наблюдающий контроллер).

Основные преимущества многослойного архитектурного стиля

– Абстракция. Уровни обеспечивают возможность внесения изменений на абстрактном уровне. Используемый уровень абстракции каждого слоя может быть увеличен или уменьшен.
– Изоляция. Обновления технологий могут быть изолированы в отдельных слоях, что поможет сократить риск и минимизировать воздействие на всю систему.

- Управляемость. Разделение основных функций помогает идентифицировать зависимости и организовать код в секции, что повышает управляемость.
- Производительность. Распределение слоев по нескольким физическим уровням может улучшить масштабируемость, отказоустойчивость и производительность.
- Возможность повторного использования. Роли повышают возможность повторного использования. Например, в MVC Контроллер часто может использоваться повторно с другими совместимыми Представлениями для обеспечения характерного для роли или настроенного для пользователя представления одних и тех же данных и функциональности.
- Тестируемость. Улучшение тестируемости является результатом наличия строго определенных интерфейсов слоев, а также возможности переключения между разными реализациями интерфейсов слоев. Шаблоны Separated Presentation позволяют создавать во время тестирования фиктивные объекты, имитирующие поведение реальных объектов, таких как Модель, Контроллер или Представление.

Рекомендации к применению

Рассмотрите возможность применения многослойной архитектуры в следующих случаях:

- в вашем распоряжении имеются уже готовые уровни, подходящие для повторного использования в других приложения;
- имеются приложения, предоставляющие подходящие бизнес- процессы через интерфейсы сервисов;
- создается сложное приложение и предварительное проектирование требует разделения, чтобы группы могли сосредоточиться на разных участках функциональности.

Многослойная архитектура также будет уместна, если приложение должно поддерживать разные типы клиентов и разные устройства, или если требуется реализовать сложные и/или настраиваемые бизнес-правила и процессы.

Архитектура, основанная на шине сообщений

Основанная на шине сообщений архитектура описывает принцип использования программной системы, которая может принимать и отправлять сообщения по одному или более каналам связи, обеспечивая, таким образом, приложениям возможность взаимодействия без необходимости знания конкретных деталей друг о друге.

Это стиль проектирования, в котором взаимодействия между приложениями осуществляются путем передачи (обычно асинхронной) сообщений через общую шину. В типовых реализациях архитектуры, основанной на шине сообщений, используется либо маршрутизатор сообщений, либо шаблон Publish/Subscribe (Публикация/Подписка) и система обмена сообщениями, такая как Message Queuing (Очередь сообщений). Многие реализации состоят из отдельных приложений, обмен данными между которыми

осуществляется путем отправки и приема сообщений по общим схемам и инфраструктуре.

Шина сообщений обеспечивает возможность обрабатывать:

– Основанное на сообщениях взаимодействие.

Все взаимодействие между приложениями основывается на сообщениях, использующих известные схемы.

– Сложную логику обработки.

Сложные операции могут выполняться как часть многошагового процесса путем сочетания ряда меньших операций, каждая из которых поддерживает определенные задачи.

– Изменения логики обработки.

Взаимодействие с шиной реализуется по общим схемам и с применением обычных команд, что обеспечивает возможность вставки или удаления приложений на шине для изменения используемой для обработки сообщений логики.

– Интеграцию с разными инфраструктурами.

Использование модели связи посредством сообщений, основанной на общих стандартах, позволяет взаимодействовать с приложениями, разработанными для разных инфраструктур, таких как Microsoft .NET и Java.

Шины сообщений используются для обеспечения сложных правил обработки. Такой дизайн обеспечивает подключаемую архитектуру, которая позволяет вводить приложения в процесс или улучшать масштабируемость, подключая к шине несколько экземпляров одного и того же приложения.

К разновидностям шины сообщений относятся:

– Сервисная шина предприятия (Enterprise Service Bus, ESB).

ESB основывается на шине сообщений и использует сервисы для обмена данными между шиной и компонентами, подключенными к шине. Обычно ESB обеспечивает сервисы для преобразования одного формата в другой, обеспечивая возможность связи между клиентами, использующими несовместимые форматы сообщений.

– Шина Интернет-сервисов (Internet Service Bus, ISB).

Подобна сервисной шине предприятия, но приложения размещаются не в сети предприятия, а в облаке. Основная идея ISB - использование Унифицированных идентификаторов ресурсов (Uniform Resource Identifiers, URIs) и политик, управляющих логикой маршрутизации через приложения и сервисы в облаке.

Основные преимущества архитектуры, основанной на шине сообщений

– Расширяемость. Возможность добавлять или удалять приложения с шины без влияния на существующие приложения.
– Невысокая сложность. Приложения упрощаются, потому что каждому из них необходимо знать лишь, как обмениваться данными с шиной.
– Гибкость. Приведение набора приложений, составляющих сложный процесс, или схем связи между приложениями в соответствие

изменяющимся бизнес-требованиям или требованиям пользователя просто путем внесения изменений в конфигурацию или параметры, управляющие маршрутизацией.

- Слабое связывание. Кроме предоставляемого приложением интерфейса для связи с шиной сообщений, нет никаких других зависимостей с самим приложением, что обеспечивает возможность изменения, обновления и замены его другим приложением, предоставляющим такой же интерфейс.
- Масштабируемость. Возможность подключения к шине множества экземпляров одного приложения для обеспечения одновременной обработки множества запросов.
- Простота приложения. Несмотря на то, что реализация шины сообщений усложняет инфраструктуру, каждому приложению приходится поддерживать лишь одно подключение к шине сообщений, а не множество подключений к другим приложениям.

Рекомендации к применению

Рассмотрите возможность применения архитектуры основанной на шине сообщений в следующих случаях:

- имеете существующие приложения, выполняющие задачи путем взаимодействия друг с другом;
- хотите объединить множество шагов в одну операцию.

Такой архитектурный стиль также подойдет при реализации решений, требующих взаимодействия с внешними приложениями или приложениями, размещенными в разных средах.

N-уровневая / 3-уровневая архитектура

N-уровневая и 3-уровневая архитектура являются стилями развертывания, описывающими разделение функциональности на сегменты, во многом аналогично многослойной архитектуре, но в данном случае эти сегменты могут физически размещаться на разных компьютерах, их называют уровнями.

Данные архитектурные стили были созданы на базе компонентно-ориентированного подхода и, как правило, для связи используют методы платформы, а не сообщения.

Характеристиками N-уровневой архитектуры приложения являются функциональная декомпозиция приложения, сервисные компоненты и их распределенное развертывание, что обеспечивает повышенную масштабируемость, доступность, управляемость и эффективность использования ресурсов. Каждый уровень абсолютно независим от всех остальных, кроме тех, с которыми он непосредственно соседствует. N-ному уровню требуется лишь знать, как обрабатывать запрос от n+1 уровня, как передавать этот запрос на n-1 уровень (если таковой имеется), и как обрабатывать результаты запроса. Для обеспечения лучшей масштабируемости связь между уровнями обычно асинхронная.

N-уровневая архитектура обычно имеет не менее трех отдельных логических частей, каждая из которых физически размещается на разных

серверах. Каждая часть отвечает за определенную функциональность. При использовании многослойного подхода слой развертывается на уровне, если предоставляемая этим слоем функциональность используется более чем одним сервисом или приложением уровня.

Примером N-уровневого/3-уровневого архитектурного стиля может служить типовое финансовое Веб-приложение с высокими требованиями к безопасности. Бизнес-слой в этом случае должен быть развернут за межсетевым экраном, из-за чего приходится развертывать слой представления на отдельном сервере в пограничной сети. Другой пример - типовой насыщенный клиент, в котором слой представления развернут на клиентских компьютерах, а бизнес-слой и слой доступа к данным развернуты на одном или более серверных уровнях.

Основные преимущества N-уровневого/3-уровневого архитектурного стиля

- Удобство поддержки. Уровни не зависят друг от друга, что позволяет выполнять обновления или изменения, не оказывая влияния на приложение в целом.
- Масштабируемость. Уровни организовываются на основании развертывания слоев, поэтому масштабировать приложение довольно просто.
- Гибкость. Управление и масштабирование каждого уровня может выполняться независимо, что обеспечивает повышение гибкости.
- Доступность. Приложения могут использовать модульную архитектуру, которая позволяет использовать в системе легко масштабируемые компоненты, что повышает доступность.

Рекомендации к применению

Рассмотрите возможность применения N-уровневой или 3-уровневой архитектуры в следующих случаях:

- требования по обработке уровней приложения отличаются настолько сильно, что может возникнуть перекос в распределении ресурсов, или существенно разнятся требования по безопасности уровней. Например, конфиденциальные данные не должны храниться на уровне представления, но могут размещаться на бизнес-уровне или уровне данных. N-уровневая или 3-уровневая архитектура также подойдет в случае;
- требуется обеспечить возможность совместного использования бизнес-логики разными приложениями и имеется достаточное количество оборудования для выделения необходимого числа серверов для каждого уровня.

Используйте только три уровня, если создаете приложение для внутренней сети организации, где все серверы будут располагаться в закрытой сети, или Интернет-приложение, требования по безопасности которого не запрещают развертывание бизнес-логики на Веб-сервере или сервере приложений.

Рассмотрите возможность применения более трех уровней, если соответственно требованиям по безопасности бизнес-логика не может быть развернута в пограничной сети, или если приложение интенсивно использует ресурсы, и для разгрузки сервера необходимо перенести эту функциональность на другой сервер.

Объектно-ориентированная архитектура

Объектно-ориентированная архитектура [9] – это парадигма проектирования, основанная на разделении ответственностей приложения или системы на самостоятельные пригодные для повторного использования объекты, каждый из которых содержит данные и поведение, относящиеся к этому объекту.

При объектно-ориентированном проектировании система рассматривается не как набор подпрограмм и процедурных команд, а как наборы взаимодействующих объектов. Объекты обособлены, независимы и слабо связаны; обмен данными между ними происходит через интерфейсы путем вызова методов и свойств других объектов и отправки/приема сообщений.

Основными принципами объектно-ориентированного архитектурного стиля являются:

– Абстракция.

Позволяет преобразовать сложную операцию в обобщение, сохраняющее основные характеристики операции. Например, абстрактный интерфейс может быть широко известным описанием, поддерживающим операции доступа к данным через использование простых методов, таких как Get (Получить) и Update (Обновить). Другая форма абстракции - метаданные, используемые для обеспечения сопоставления двух форматов структурированных данных.

– Композиция.

Объекты могут быть образованы другими объектами и по желанию могут скрывать эти внутренние объекты от других классов или предоставлять их как простые интерфейсы.

– Наследование.

Объекты могут наследоваться от других объектов и использовать функциональность базового объекта или переопределять ее для реализации нового поведения. Более того, наследование упрощает обслуживание и обновление, поскольку изменения, вносимые в базовый объект, автоматически распространяются на все наследуемые от него объекты.

– Инкапсуляция.

Объекты предоставляют функциональность только через методы, свойства и события и скрывают внутренние детали, такие как состояние и переменные, от других объектов. Это упрощает обновление или замену объектов и позволяет выполнять эти операции без влияния на другие объекты и код, требуется лишь обеспечить совместимые интерфейсы.

– Полиморфизм.

Позволяет переопределять поведение базового типа, поддерживающего операции в приложении, путем реализации новых типов, которые являются взаимозаменяемыми для существующего объекта.

Обычно объектно-ориентированный стиль используется для описания объектной модели, поддерживающей сложные научные или финансовые операции, либо описания объектов, представляющих реальные артефакты предметной области (такие как покупатель или заказ). Последнее чаще реализуется с применением более специализированного стиля проектирования на основе предметной области, который использует преимущества принципов объектно-ориентированной архитектуры.

Основные преимущества объектно-ориентированной архитектуры

- Понятность. Обеспечивается более близкое соответствие приложения реальным объектам, что делает его более понятным.
- Возможность повторного использования. Обеспечивается возможность повторного использования через полиморфизм и абстракцию.
- Тестируемость. Обеспечивается улучшенная тестируемость через инкапсуляцию.
- Расширяемость. Инкапсуляция, полиморфизм и абстракция гарантируют, что изменения в представлении данных не повлияют на интерфейсы, предоставляемые объектами, что могло бы ограничить возможности связи и взаимодействия с другими объектами.
- Высокая связность. Размещая в объекте только функционально близкие методы и функции и используя для разных наборов функций разные объекты, можно достичь высокого уровня связности.

Рекомендации к применению

Рассмотрите возможность применения объектно-ориентированной архитектуры в следующих случаях:

- моделирование приложения на базе реальных объектов и действий;
- имеются подходящие объекты и классы, соответствующие проектным и эксплуатационным требованиям.

Объектно-ориентированный стиль также подойдет, если необходимо инкапсулировать логику и данные в компоненты, пригодные для повторного использования, или если имеется сложная бизнес-логика, которая требует абстракции и динамического поведения.

Сервисно-ориентированная архитектура

Сервисно-ориентированная архитектура (Service-oriented architecture, SOA) обеспечивает возможность предоставлять функциональность приложения в виде набора сервисов и создавать приложения, использующие программные сервисы.

Сервисы слабо связаны, потому что используют основанные на стандартах интерфейсы, которые могут быть вызваны, опубликованы и обнаружены. Основная задача сервисов в SOA – предоставление схемы и взаимодействия с приложением посредством сообщений через интерфейсы, областью действия

которых является приложение, а не компонент или объект. Не следует рассматривать SOA-сервис как компонентный поставщик сервисов.

SOA-архитектура может обеспечить упаковку бизнес-процессов в сервисы, поддерживающие возможность взаимодействия и использующие для передачи информации широкий диапазон протоколов и форматов данных. Клиенты и другие сервисы могут выполнять доступ к локальным сервисам, выполняющимся на том же уровне, или к удаленным сервисам по сети.

Основными принципами архитектурного стиля SOA являются:

- Сервисы автономны. Обслуживание, разработка, развертывание и контроль версий каждого сервиса происходит независимо от других.
- Сервисы могут быть распределены. Сервисы могут размещаться в любом месте сети, локально или удаленно, если сеть поддерживает необходимые протоколы связи.
- Сервисы слабо связаны. Каждый сервис совершенно не зависит от остальных и может быть заменен или обновлен без влияния на приложения, его использующие, при условии предоставления совместимого интерфейса.
- Сервисы совместно используют схему и контракт, но не класс. При обмене данными сервисы совместно используют контракты и схемы, но не внутренние классы.
- Совместимость основана на политике. Политика, в данном случае, означает описание характеристик, таких как транспорт, протокол и безопасность.

Типовые сервисно-ориентированные приложения обеспечивают совместное использование информации, выполнение многоэтапных процессов (системы резервирования и онлайн- магазины), предоставление специальных отраслевых данных или сервисов между организациями и создание составных приложений, которые объединяют данные из многих источников.

Основные преимуществами SOA-архитектуры

- Согласование предметных областей. Повторное использование общих сервисов со стандартными интерфейсами расширяет технологические и бизнес-возможности, а также сокращает стоимость.
- Абстракция. Сервисы являются автономными, доступ к ним осуществляется по формальному контракту, что обеспечивает слабое связывание и абстракцию.
- Возможность обнаружения. Сервисы могут предоставлять описания, что позволяет другим приложениям и сервисам обнаруживать их и автоматически определять интерфейс.
- Возможность взаимодействия. Поскольку протоколы и форматы данных основываются на отраслевых стандартах, поставщик и потребитель сервиса могут создаваться и развертываться на разных платформах.
- Рационализация. Сервисы обеспечивают определенную функциональность, устраняя необходимость ее дублирования в приложениях.

Рекомендации к применению

Рассмотрите возможность применения SOA-подхода в следующих случаях:

- имеется доступ к сервисам, которые желаете повторно использовать;
- есть возможность приобрести подходящие сервисы у компании, предоставляющей услуги хостинга;
- требуется создать приложения, объединяющие различные сервисы в один UI;
- требуется создать приложения в модели ПО + сервисы (Software plus Services, S+S), ПО как сервис (SaaS) или приложения для размещения в облаке.

SOA-стиль также подходит, если требуется поддерживать связь посредством обмена сообщениями между сегментами приложения и предоставлять функциональность независимо от платформы, если хотите использовать интегрированные сервисы, такие как аутентификация, или хотите предоставлять сервисы, видимые в каталогах, с возможностью использования клиентами, которые заранее ничего не знают об интерфейсах.

1.4 Методика построения архитектуры и дизайна[4]

В данном вопросе описывается итеративная методика, которая может использоваться при создании прототипа будущей архитектуры.

Данная методика позволит свести воедино ключевые решения по параметрам качества, архитектурным стилям, типам приложений, технологиям и сценариям развертывания.

Методика предполагает создание архитектуры в ходе процесса, состоящего из серий по пять основных шагов. В свою очередь, каждый шаг разбит на отдельные аспекты.

Итеративный процесс помогает выработать возможные варианты решений, которые в дальнейшем дорабатываются в ходе итераций и, в конечном счете, обеспечивают создание дизайна архитектуры, наиболее соответствующей разрабатываемому приложению. В конце процесса можно создать обзор архитектуры и представить его всем заинтересованным сторонам.

В зависимости от подхода, используемого вашей организацией для разработки ПО, архитектура может многократно пересматриваться в ходе жизненного цикла проекта. Эта методика подходит для дальнейшей доработки архитектуры, дополнения ее новыми аспектами, выявленными в последующий период сбора сведений, создания прототипов и фактической разработки.

Тем не менее, важно понимать, что это всего лишь один из возможных подходов. Существует множество других более формальных методов определения, анализа и представления архитектуры.

---

[4] Практическое занятие №1.

Исходные данные, выходные данные и этапы проектирования

Исходные данные проектирования помогают формализовать требования и ограничения, которые должна реализовать создаваемая архитектура.

Обычно исходными данными являются варианты использования и сценарии поведения пользователя, функциональные требования, нефункциональные требования (включая параметры качества, такие как производительность, безопасность, надежность и другие), технологические требования, целевая среда развертывания и другие ограничения.

В ходе процесса разработки создается список значимых с точки зрения архитектуры вариантов использования, аспектов архитектуры, требующих специального внимания, и возможных архитектурных решений, которые удовлетворяют требованиям и ограничениям, выявленным в процессе проектирования.

Общей техникой постепенной доработки дизайна до тех пор, пока он не будет удовлетворять всем требованиям и ограничениям, является итеративная методика, включающая пять основных этапов, как показано на рис. 1.3.

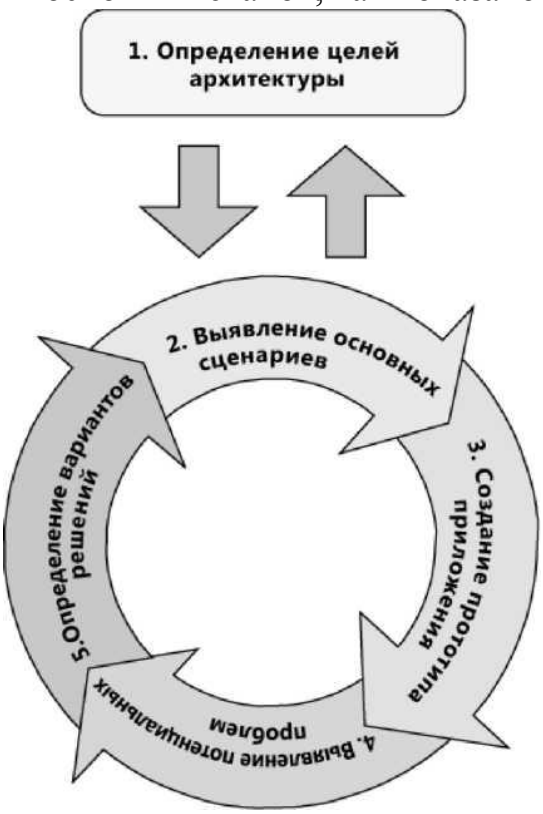

Рис. 1.3 Основные этапы итеративного процесса проектирования архитектуры

Основные этапы итеративного процесса:

1. Определение целей архитектуры.

Наличие четких целей поможет сосредоточиться на архитектуре и правильном выборе проблем для решения. Точно обозначенные цели помогают определить границы каждой фазы: момент, когда завершена текущая фаза и все готово для перехода к следующей.

2. Основные сценарии.

Используйте основные сценарии, чтобы сосредоточиться на том, что имеет первостепенное значение, и проверяйте возможные варианты архитектур на соответствие этим сценариям.

3. Общее представление приложения.

Определите тип приложения, архитектуру развертывания, архитектурные стили и технологии, чтобы обеспечить соответствие вашего дизайна реальным условиям, в которых будет функционировать создаваемое приложение.

4. Потенциальные проблемы.

Выявите основные проблемные области на основании параметров качества и потребности в сквозной функциональности. Это области, в которых чаще всего делаются ошибки при проектировании приложения.

5. Варианты решений.

В каждой итерации должен быть создан прототип архитектуры, являющийся развитием и доработкой решения. Прежде чем переходить к следующей итерации, необходимо убедиться в соответствии этого прототипа основным сценариям, проблемам и ограничениям развертывания.

Такой процесс создания архитектуры предполагает итеративный и инкрементный подход.

Сначала создается возможный вариант архитектуры – обобщенный дизайн, который может тестироваться по основным сценариям, требованиям, известным ограничениям, параметрам качества и коллекцией архитектурных аспектов на которые нужно акцентировать внимание.

В ходе доработки варианта архитектуры, выявляются дополнительные детали и сведения о дизайне, результатом чего становится расширение основных сценариев, корректировка общего представления приложения и подхода к решению проблем.

При итеративном походе к архитектуре часто есть соблазн выполнять итерации в горизонтальном направлении, в рамках отдельных слоев приложения, а не в вертикальном направлении, что заставляет думать о функциональности, выходящей за рамки слоев и составляющей отдельную возможность (вариант использования) значимую для пользователей. При выполнении итераций в горизонтальной плоскости есть угроза реализации большого числа функций до того, как пользователи смогут их проверить.

Не стремитесь создать архитектуру за одну итерацию. Каждая итерация должна раскрывать дополнительные детали. Но не увязайте в деталях, сосредоточьтесь на основных этапах и создавайте инфраструктуру, на которой может быть основана ваша архитектура и дизайн. В следующих разделах предлагаются рекомендации и сведения по каждому из этапов.

Определение целей архитектуры

Цели архитектуры – это задачи и ограничения, очерчивающие архитектуру и процесс проектирования, определяющие объем работ и помогающие понять, когда пора остановиться.

Рассмотрим ключевые моменты в определении целей архитектуры:

- Начальное определение задач архитектуры.

От этих задач будет зависеть время, затрачиваемое на каждую фазу проектирования архитектуры. Необходимо решить, что вы делаете: создаете прототип, проводите тестирование возможных вариантов реализации или выполняете длительный процесс разработки архитектуры для нового приложения.

– Определение потребителей архитектуры.

Определите, будет ли разрабатываемая конструкция использоваться другими архитекторами, либо она предназначается для разработчиков и тестировщиков, ИТ-специалистов и руководителей. Учтите нужды и подготовленность целевой аудитории, чтобы сделать разрабатываемую конструкцию максимально удобной для них.

– Определение ограничений.

Изучите все опции и ограничения применяемой технологии, ограничения использования и развертывания. Полностью разберитесь со всеми ограничениями в начале работы, чтобы не тратить время или не сталкиваться с сюрпризами в процессе разработки приложения.

Время и объем работ

На основании общих целей архитектуры можно планировать время, затрачиваемое на каждую из работ по проектированию. Например, на разработку прототипа может уйти лишь несколько дней, тогда как для создания полной и детально проработанной архитектуры сложного приложения потребуются месяцы и множество итераций. Исходя из своего понимания целей, оценивайте необходимое количество времени и сил для каждого этапа, это поможет прийти к видению результата и четкому определению целей и приоритетов архитектуры.

Перечислим возможные цели:

– создание полного дизайна приложения;
– создание прототипа;
– определение основных технических рисков;
– тестирование возможных вариантов реализации;
– создание общих моделей для облегчения понимания системы.

Ключевые сценарии

В контексте архитектуры вариант использования (use case) – это описание ряда взаимодействий между системой и одним или более действующими лицами (либо пользователем, либо другой системой).

Сценарий – это более широкое и всеобъемлющее описание взаимодействия пользователя с системой, чем ветвь варианта использования. Основной целью при продумывании архитектуры системы должно быть выявление нескольких ключевых сценариев, что поможет при принятии решения об архитектуре.

Задача состоит в том, чтобы найти баланс между целями пользователя, бизнеса и системы.

Ключевые сценарии – это наиболее важные сценарии для успеха создаваемого приложения. Ключевой сценарий можно определить как любой сценарий, отвечающий одному или более из следующих критериев:

- представляет проблемную область – значительную неизвестную область или область значительного риска;
- ссылается на существенный для архитектуры вариант использования.
- представляет взаимодействие параметров качества с функциональностью.
- представляет компромисс между параметрами качества.

Например, сценарии аутентификации пользователей могут быть ключевыми сценариями, потому что являются пересечением параметра качества (безопасность) с важной функциональностью (регистрация пользователя в системе). В качестве другого примера можно привести сценарий, основанный на незнакомой или новой технологии.

Важные с точки зрения архитектуры варианты использования

Важные с точки зрения архитектуры варианты использования оказывают влияние на многие аспекты дизайна. Они играют особо важную роль в обеспечении будущего успеха создаваемого приложения. Эти варианты использования важны для приемки развернутого приложения и должны охватывать достаточно большую часть дизайна, чтобы быть полезными при оценке архитектуры.

К важным с точки зрения архитектуры вариантам использования относятся:

- Бизнес-критический (Business Critical).

Вариант использования, имеющий высокий уровень использования либо особую важность для пользователей или других заинтересованных сторон, по сравнению с другими функциями, или предполагающий высокий риск.

- Имеющий большое влияние (High Impact).

Вариант использования охватывает и функциональность, и параметры качества, либо представляет сквозную функцию, имеющую глобальное влияние на слои и уровни приложения. Примерами могут служить особо уязвимые с точки зрения безопасности операции Create, Read, Update, Delete (CRUD).

После выявления важных с точки зрения архитектуры вариантов использования они могут применяться как средство оценки применимости или неприменимости возможных вариантов архитектуры приложения. Если вариант архитектуры охватывает больше вариантов использования или описывает существующие варианты использования более эффективно, обычно это свидетельствует о том, что данный вариант архитектуры является улучшением базовой архитектуры.

Хороший вариант использования будет совпадать с пользовательским представлением, системным представлением и бизнес-представлением архитектуры. Используйте эти сценарии и варианты использования для тестирования своего дизайна и выявления возможных проблем.

При продумывании вариантов использования и сценариев обратите внимание на следующее:

- на ранних этапах разработки дизайна сократите риск путем создания варианта архитектуры, поддерживающего важные с точки зрения архитектуры сквозные сценарии, затрагивающие все слои архитектуры;
- используя модель архитектуры как руководство, вносите изменения в архитектуру, дизайн и код для реализации сценариев, функциональных требований, технологических требований, параметров качества и ограничений;
- создайте модель архитектуры на основании известных на данный момент сведений и составьте список вопросов, ответы на которые должны быть даны в последующих сценариях и итерациях;
- внеся существенные изменения в архитектуру и дизайн, создайте вариант использования, который будет отражать и применять эти изменения;

Общее представление приложения

Общее представление позволит сделать архитектуру более осязаемой, свяжет ее с реальными ограничениями и решениями.

Создание общего представления приложения включает следующие действия:

1. Определение типа приложения.

Прежде всего, определите, приложение какого типа создается, например, мобильное приложение, насыщенный клиент, насыщенное Интернет-приложение, сервис, Веб-приложение или некоторое сочетание этих типов.

2. Определение ограничений развертывания.

При проектировании архитектуры приложения необходимо учесть корпоративные политики и процедуры, а также среду, в которой планируется развертывание приложения. Если целевая среда фиксированная или негибкая, конструкция приложения должна отражать существующие в этой среде ограничения. Также в конструкции приложения должны быть учтены нефункциональные требования (Quality-of-Service, QoS), такие как безопасность и надежность. Иногда необходимо поступиться чем-либо в дизайне из-за ограничений в поддерживаемых протоколах или топологии сети. Выявление требований и ограничений, присутствующих между архитектурой приложения и архитектурой среды на ранних этапах проектирования позволяет выбрать соответствующую топологию развертывания и разрешить конфликты между приложением и целевой средой.

3. Определение значащих архитектурных стилей проектирования.

Определите, какие архитектурные стили будут использоваться при проектировании. Стиль может рассматриваться как обобщенный шаблон, обеспечивающий абстрактную базу для семейства систем. Каждый стиль определяет набор правил, задающих типы компонентов, которые могут использоваться для компоновки системы, типы отношений, применяемых в компоновке, ограничения по способам компоновки и допущения о семантике компоновки. Архитектурный стиль улучшает секционирование и способствует

возможности повторного использования дизайна благодаря предоставлению решений часто встречающихся проблем. Типовым архитектурным стилям был посвящен вопрос 1.3.

4. Выбор подходящих технологий.

Наконец, на основании типа приложения и других ограничений выбираем подходящие технологии и определяем, какие технологии будут использоваться в будущей системе. Основными факторами являются тип разрабатываемого приложения, предполагаемая топология развертывания приложения и предпочтительные архитектурные стили. Выбор технологий также зависит от политик организации, ограничений среды, квалификации штата и т.д. В следующем разделе описываются некоторые основные технологии Microsoft для каждого типа приложения.

Подходящие технологии

При выборе технологий для использования при проектировании обращайте внимание на то, что обеспечит выбранный архитектурный стиль, тип и основные параметры качества для приложения. Рассмотрим рекомендации, которые помогут выбрать технологии представления, реализации и связи, наиболее подходящие для каждого типа приложений на платформе Microsoft:

– Мобильные приложения.

Для разработки приложения для мобильных устройств могут использоваться технологии слоя представления, такие как .NET Compact Framework, ASP.NET для мобильных устройств и Silverlight для мобильных устройств.

– Насыщенные клиентские приложения.

Для разработки приложений с насыщенными UI, развертываемыми и выполняемыми на клиенте, могут использоваться сочетания технологий слоя представления Windows Presentation Foundation (WPF), Windows Forms и XAML Browser Application (XBAP).

– Насыщенные клиентские Интернет-приложения (RIA).

Для развертывания насыщенных UI в рамках Веб-браузера могут использоваться подключаемый модуль Silverlight™ или Silverlight в сочетании с AJAX.

– Веб-приложения.

Для создания Веб-приложений могут применяться ASP.NET Web Forms, AJAX, Silverlight, ASP.NET MVC и ASP.NET Dynamic Data.

– Сервисные приложения.

Для создания сервисов, предоставляющих функциональность внешним потребителям систем и сервисов, могут использоваться Windows Communication Foundation (WCF) и ASP.NET Web services (ASMX).

Графическое представление архитектуры

Важно, чтобы вы могли графически представить разрабатываемую архитектуру. Независимо от того, делается ли это на бумаге, в виде слайдов или в другом формате, важно показать основные ограничения и принятые решения для того, чтобы обозначить границы и начать обсуждение. На самом деле, это имеет

двойную ценность. Если вы не можете наглядно представить архитектуру, значит, вы не полностью понимаете ее. Если вы смогли изобразить четкую и краткую диаграмму, она будет понятной остальным, и будет намного проще объяснять детали.

На рисунке 1.4 показан пример графического представления дизайна Веб-приложения в первом приближении с указанием протоколов и методов аутентификации, которые предполагается использовать.

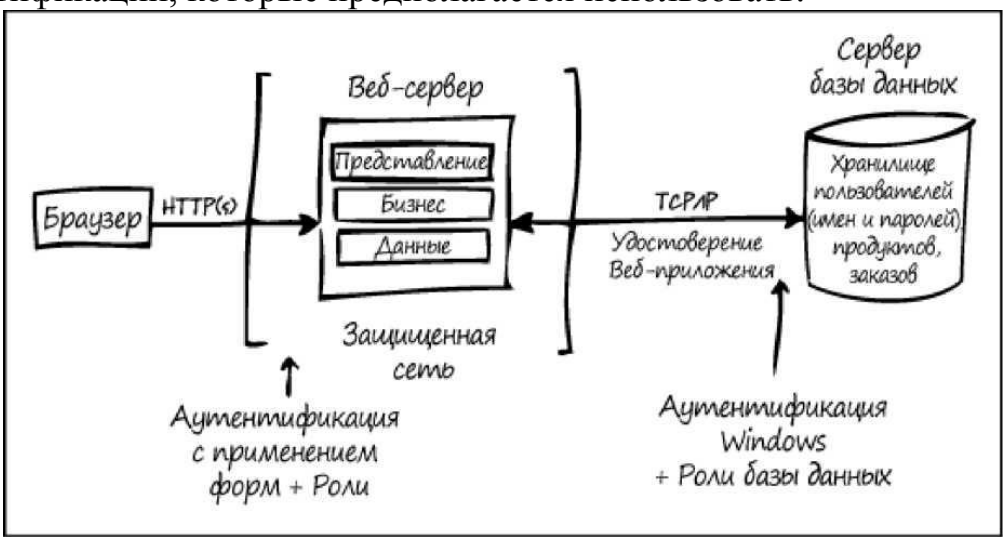

Рис. 1.4 Пример графического представления дизайна Веб-приложения

Потенциальные проблемы

Определите основные потенциальные проблемы архитектуры своего приложения, чтобы понять области, в которых наиболее вероятно возникновение ошибок.

К потенциальным проблемам относятся появление новых технологий и критически важные бизнес-требования. Например, «Могу ли я переходить с одного сервиса стороннего производителя к другому?», «Могу ли я добавить поддержку нового типа клиента?», «Могу ли я быстро менять бизнес- правила оплаты услуг?» и «Могу ли я перейти к новой технологии для компонента X?». Несмотря на то, что это крайне обобщенные аспекты, как правило, при реализации они (и другие зоны риска) проецируются в параметры качества и сквозную функциональность.

Вопросы, требующие особого внимания при проектировании

Параметры качества

В пункте 1.2 было сказано о необходимости оценивать качество создаваемой архитектуры с помощью свойств, которые оказывают влияние на поведение во время выполнения, дизайн системы и взаимодействие с пользователем. Та степень, с которой приложение обеспечивает требуемое сочетание параметров качества, таких как удобство и простота использования, производительность, надежность и безопасность, определяет успешность дизайна и общее качество программного продукта. При проектировании приложения, отвечающего любому из этих параметров, необходимо учесть влияние и других требований, должны быть проанализированы плюсы и минусы

по отношению к другим параметрам качества. Важность или приоритетность каждого из параметров качества для разных систем разная. Например, для бизнес-приложения (line-of-business, LOB) производительность, масштабируемость, безопасность и удобство использования будут более важны, чем возможность взаимодействия с другими системами. А вот для коробочного приложения такая возможность будет иметь большее значение, чем для LOB-приложения.

Параметры качества представляют функциональные области, которые потенциально могут оказывать влияние на все приложение, на все его слои и уровни. Некоторые параметры относятся ко всему дизайну системы, тогда как другие касаются только времени выполнения, времени проектирования или взаимодействия с пользователем. Более подробно показатели качества описываются в пункте 2.9.

Сквозная функциональность

Анализ параметров качества и сквозной функциональности в связи с имеющимися требованиями позволяет сосредоточиться на конкретных функциональных областях.

Например, безопасность, несомненно, является жизненно важным фактором при проектировании и присутствует во многих слоях и во многих аспектах архитектуры. Сквозная функциональность, относящаяся к безопасности, является ориентиром, указывающим на области, на которых следует заострить внимание. Категории сквозной функциональности могут использоваться для разделения архитектуры приложения для дальнейшего анализа и выявления уязвимых мест приложения. Такой подход обеспечивает создание дизайна с оптимальным уровнем безопасности.

При обсуждении сквозной функциональности относящейся к безопасности рекомендуется обратить внимание на следующие вопросы:

– Аудит и протоколирование.

Кто, что сделал и когда? Приложение функционирует в нормальном режиме? Аудит занимается вопросами регистрации событий, связанных с безопасностью. Протоколирование касается того, как приложение публикует данные о своей работе.

– Аутентификация.

Кто вы? Аутентификация - это процесс, при котором одна сущность четко и однозначно идентифицирует другую сущность, обычно это делается с помощью таких учетных данных, как имя пользователя и пароль.

– Авторизация.

Что вы можете делать? Авторизация определяет, как приложение управляет доступом к ресурсам и операциям.

– Управление конфигурацией.

В каком контексте выполняется приложение? К каким базам данных подключается? Как выполняется администрирование приложения? Как защищены эти настройки? Управление конфигурацией определяет, как приложение реализует эти операции и задачи.

– Шифрование.

Как реализована защита секретов (конфиденциальных данных)? Как осуществляется защита от несанкционированного доступа данных и библиотек (целостности)? Как передаются случайные значения, которые должны быть криптографически стойкими? Шифрование и криптография занимается вопросами реализации конфиденциальности и целостности.

– Обработка исключений.

Что делает приложение при сбое вызова его метода? Насколько полные данные об ошибке оно предоставляет? Обеспечивает ли оно понятные для конечных пользователей сообщения об ошибках? Возвращает ли оно ценные сведения об исключении вызывающей стороне? Выполняется ли корректная обработка произошедшего сбоя? Предоставляет ли приложение администраторам необходимую информацию для проведения анализа основных причин сбоя? Обработка исключений касается того, как исключения обрабатываются в приложении.

– Валидация входных данных.

Как определить, что поступаемые в приложение данные действительные и безопасные? Выполняется ли ограничение ввода через точки входа и кодировка вывода через точки выхода? Можно ли доверять таким источникам данных, как базы данных и общие файлы? Проверка ввода касается вопросов фильтрации, очистки или отклонения вводимых в приложение данных перед их дополнительной обработкой.

– Конфиденциальные данные.

Как приложение работает с конфиденциальными данными? Обеспечивает ли оно защиту конфиденциальных данных пользователей и приложения? Здесь решаются вопросы обработки приложением любых данных, которые должны быть защищены либо при хранении в памяти, либо при передаче по сети, либо при хранении в постоянных хранилищах.

– Управление сеансами.

– Как приложение обрабатывает и защищает сеансы пользователей?

Эти вопросы помогут принять основные проектные решения по безопасности приложения и задокументировать их как часть архитектуры. Например, на рис. 1.5 показано, как аспекты безопасности обозначены в архитектуре типового Веб-приложения.

Варианты решений

Определив основные проблемы, можно приступать к созданию исходной базовой архитектуры и ее детализации для получения возможного варианта архитектуры. В ходе этого процесса можно использовать пилотные архитектуры для более подробного рассмотрения определенных областей дизайна или для проверки новых идей. Затем выполняется проверка нового варианта архитектуры на соответствие основным сценариям и заданным требованиям и вновь повторяется итеративный цикл по доработке дизайна.

Рис. 1.5 Аспекты безопасности в архитектуре типового Веб-приложения

Важно, особенно если проектирование и разработка ведутся по гибкому процессу, чтобы итерация включала как по проектирование архитектуры, так и по разработку реализации. Это поможет избежать масштабного проектирования наперед.

Базовая архитектура и возможные варианты архитектуры

Базовая архитектура описывает существующую систему, то, как она выглядит сегодня. Для нового проекта исходная базовая архитектура - это первое высокоуровневое представление архитектуры, на основании которого будут создаваться возможные варианты архитектуры. Возможный вариант архитектуры включает тип приложения, архитектуру развертывания, архитектурный стиль, выбранные технологии, параметры качества и сквозную функциональность.

На каждом этапе разработки дизайна будьте уверены, что понимаете основные риски и предпринимаете меры по их сокращению, проводите оптимизацию для эффективной и рациональной передачи проектных сведений и создаете архитектуру, обеспечивая гибкость и возможность реструктуризации. Возможно, архитектуру придется изменять несколько раз, использовать несколько итераций, возможных вариантов и множество пилотных архитектур.

Если возможный вариант архитектуры является улучшением, он может стать базой для создания и тестирования новых возможных вариантов.

Итеративный и инкрементный подход позволяет избавиться от крупных рисков сначала, итеративно формировать архитектуру и через тестирование подтверждать, что каждая новая базовая архитектура является улучшением предыдущей.

Следующие вопросы помогут протестировать новый вариант архитектуры, полученный на основании «пилота» архитектуры:

Данная архитектура обеспечивает решение без добавления новых рисков?

Данная архитектура устраняет больше известных рисков, чем предыдущая итерация?

Данная архитектура реализует дополнительные требования?

Данная архитектура реализует важные с точки зрения архитектуры варианты использования?

Данная архитектура реализует аспекты, связанные с параметрами качества?

Данная архитектура реализует дополнительные аспекты сквозной функциональности?

Пилотные архитектуры

Пилотная архитектура (architectural spike) – это тестовая реализация небольшой части общего дизайна или архитектуры приложения.

Ее назначение – анализ технических аспектов конкретной части решения для проверки технических допущений, выбора дизайна из ряда возможных вариантов и стратегий реализации или иногда оценка сроков реализации.

Пилотные архитектуры часто применяются в процессах гибкого или экстремального проектирования, но могут быть очень эффективным способом улучшения и доработки дизайна решения независимо от подхода к разработке. Благодаря их сфокусированности на основных частях общего проекта решения, пилотные архитектуры могут использоваться для решения важных технических проблем и для сокращения общих рисков и неопределенностей в дизайне.

После завершения моделирования архитектуры можно приступать к доработке дизайна, планированию тестов и представлению решений остальным участникам процесса, при этом руководствуйтесь следующими рекомендациями.

При документировании возможных вариантов архитектуры и вариантов ее тестирования старайтесь не загромождать этот документ, что обеспечит простоту его обновления. Такой документ может включать сведения о целях, типе приложения, топологии развертывания, основных сценариях и требованиях, технологиях, параметрах качества и тестах.

Используйте параметры качества для определения очертаний дизайна и реализации. Например, разработчики должны знать антишаблоны для выявленных архитектурных рисков и использовать соответствующие проверенные схемы для решения данных проблем.

Делитесь получаемыми сведениями с участниками группы и другими заинтересованными сторонами. К ним могут относиться группа разработки приложения, группа тестирования и администраторы сети или системные администраторы.

Анализ архитектуры

Анализ архитектуры приложения – критически важная задача, поскольку позволяет сократить затраты на исправление ошибок, как можно раньше выявить и исправить возможные проблемы. Анализ архитектуры следует выполнять часто: по завершении основных этапов проекта и в ответ на существенные изменения в архитектуре. Создавайте архитектуру, помня об

основных вопросах, задаваемых при таком анализе, это позволит как улучшить архитектуру, так и сократить время, затрачиваемое на каждый анализ.

Основная цель анализа архитектуры – подтверждение применимости базовой архитектуры и ее возможных вариантов, и также проверка соответствия предлагаемых технических решений функциональным требованиям и параметрам качества. Кроме того, анализ помогает обнаружить проблемы и выявить области, требующие доработки.

Оценки на основании сценариев

Оценки на основании сценариев – это мощный метод анализа дизайна архитектуры. При такой оценке основное внимание направлено на наиболее важные с точки зрения бизнеса и имеющие набольшее влияние на архитектуру сценарии. Рассмотрите возможность применения одной из следующих типовых методик:

- Метод анализа архитектуры ПО (Software Architecture Analysis Method, SAAM).

Изначально SAAM создавался для оценки модифицируемости, но позже был расширен для анализа архитектуры относительно показателей качества, таких как модифицируемость, портируемость, расширяемость, интегрируемость и функциональный охват.

- Метод анализа архитектурных компромиссов (Architecture Tradeoff Analysis Method, ATAM).

ATAM – это доработанная и улучшенная версия SAAM, которая позволяет пересматривать архитектурные решения относительно требований параметров качества и того, насколько хорошо эти решения отвечают конкретным целевым показателям качества.

- Активный анализ конструкции (Active Design Review, ADR).

ADR больше всего подходит для незавершенных архитектур или архитектур, находящихся в процессе разработки. Основное отличие этого метода в том, что анализ более сфокусирован на наборе проблем или отдельных разделах, а не на проведении общего анализа.

- Активный анализ промежуточных конструкций (Active Reviews of Intermediate Designs, ARID).

ARID сочетает в себе подход ADR анализа архитектуры, находящейся в процессе разработки, с фокусом на наборе проблем и подход методов ATAM и SAAM анализа на основании сценария с основным вниманием на параметрах качества.

- Метод анализа рентабельности (Cost Benefit Analysis Method, CBAM).

Метод CBAM основное внимание уделяет анализу затрат, выгод и планированию последствий архитектурных решений.

- Анализ модифицируемости на уровне архитектуры (Architecture Level Modifiability Analysis, ALMA).

ALMA оценивает модифицируемость архитектуры для систем бизнес-аналитики (business information systems, BIS).

– Метод оценки семейства архитектур (Family Architecture Assessment Method, FAAM).

FAAM оценивает архитектуры семейства информационных систем с точки зрения возможности взаимодействия и расширяемости.

Представление дизайна архитектуры

Представление дизайна является очень важным для проведения анализа архитектуры, также это гарантирует, что все реализовано правильно. Дизайн архитектуры должен быть представлен всем заинтересованным сторонам, включая группу разработки, системных администраторов и операторов, владельцев бизнеса и др.

Один из способов представления архитектуры – карта важных решений. Карта это не территория, а абстракция, которая помогает раскрыть и представить архитектуру.

Существует несколько широко известных методов описания архитектуры для ее представления:

– 4+1.

В данном подходе используется пять представлений готовой архитектуры. Четыре представления описывают архитектуру с разных точек зрения: логическое представление (например, объектная модель), представление процессов (например, аспекты параллелизма и синхронизации), физическое представление (схема программных уровней и функций в распределенной аппаратной среде) и представление для разработчиков. Пятое представление показывает сценарии и варианты использования ПО.

– Гибкое моделирование.

Данный подход следует идее того, что содержимое важнее представления. Это обеспечивает простоту, понятность, достаточную точность и единообразие создаваемых моделей. Простота документа гарантирует активное участие заинтересованных сторон в моделировании артефактов. Более подробно этот подход рассматривается в [15].

– IEEE 1471.

IEEE 1471 – сокращенное название стандарта, формально известного как ANSI/IEEE 1471-2000 [16], который обогащает описание архитектуры, в частности, придавая конкретное значение контексту, представлениям и срезам. Больше информации об этом можно найти в [17].

– Унифицированный язык моделирования (Unified Modeling Language, UML).

Этот подход обеспечивает три представления модели системы. Представление функциональных требований (функциональные требования системы с точки зрения пользователя, включая варианты использования); статическое структурное представление (объекты, атрибуты, отношения и операции, включая диаграммы классов) и представление динамического поведения (взаимодействие объектов и изменения внутреннего состояния объектов, включая диаграммы последовательностей, деятельностей и состояний).

Подробно об этом рассказывается в книге создателей этого языка [7].

2. Основы проектирования многослойных приложений

В данном разделе описывается общая структура приложений с точки зрения логической группировки компонентов в отдельные слои, взаимодействующие друг с другом и с другими клиентами и приложениями. Слои (layers) описывают логическую группировку функций и компонентов в приложении.

Разбиение на слои выполняется соответственно логическому делению компонентов и функциональности и не учитывает физического размещения компонентов.

Слои могут размещаться как на разных уровнях, так и на одном. Будет рассмотрено, как разделять приложения на логические части, как выбирать соответствующую функциональную компоновку приложения и как обеспечить поддержку приложением множества типов клиентов. Также будет рассказано о сервисах, которые могут использоваться для предоставления логики в слоях приложений.

Уровни (tiers) описывают физическое распределение функций и компонентов по серверам, компьютерам, сетям или удаленным местоположениям. Несмотря на то, что и для слоев, и для уровней применяется одна и так же терминология (представление, бизнес, сервисы и данные), следует помнить, что только уровни подразумевают физическое разделение. Размещение нескольких слоев на одном компьютере (одном уровне) – довольно обычное явление. Термин уровень используется в применении к схемам физического распределения, например, двухуровневое, трехуровневое, n-уровневое.

2.1 Логическое представление архитектуры многослойной системы

Независимо от типа проектируемого приложения и того, имеется ли у него пользовательский интерфейс или оно является сервисным приложением, которое просто предоставляет сервисы (не путайте со слоем сервисов приложения), его структуру можно разложить на логические группы программных компонентов. Эти логические группы называются слоями.

Слои помогают разделить разные типы задач, осуществляемые этими компонентами, что упрощает создание дизайна, поддерживающего возможность повторного использования компонентов. Каждый логический слой включает ряд отдельных типов компонентов, сгруппированных в подслои, каждый из подслоев выполняет определенный тип задач.

Определяя универсальные типы компонентов, которые присутствуют в большинстве решений, можно создать схему приложения или сервиса и затем использовать эту схему как эскиз создаваемого дизайна.

Разделение приложения на слои, выполняющие разные роли и функции, помогает максимально повысить удобство и простоту обслуживания кода, оптимизировать работу приложения при различных схемах развертывания и обеспечивает четкое разграничение областей применения определенной технологии или принятия определенных проектных решений.

Слой представления, бизнес-слой и слой данных

На самом высоком и наиболее абстрактном уровне логическое представление архитектуры системы может рассматриваться как набор взаимодействующих компонентов, сгруппированных в слои.

На рис. 2.1 показано упрощенное высокоуровневое представление этих слоев и их взаимоотношений с пользователями, другими приложениями, вызывающими сервисы, реализованные в бизнес-слое приложения, источниками данных, такими как реляционные базы данных или Веб-сервисы, обеспечивающие доступ к данным, и внешними или удаленными сервисами, используемыми приложением.

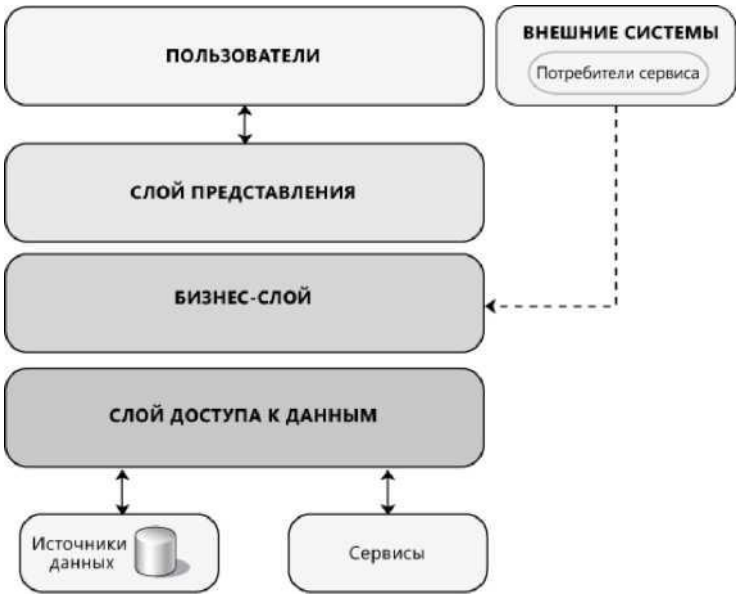

Рис. 2.1 Логическое представление архитектуры многослойной системы

Эти слои физически могут располагаться на одном или разных уровнях. Если они размещаются на разных уровнях или разделены физическими границами, дизайн должен обеспечивать это.

Как показано на рис. 2.1, приложение может состоять из ряда базовых слоев. Типовой трехслойный дизайн включает следующие слои:

– Слой представления.

Данный слой содержит ориентированную на пользователя функциональность, которая отвечает за реализацию взаимодействием пользователя с системой, и, как правило, включает компоненты, обеспечивающие общую связь с основной бизнес-логикой, инкапсулириванной в бизнес-слое.

– Бизнес-слой (слой бизнес-логики).

Этот слой реализует основную функциональность системы и инкапсулирует связанную с ней бизнес-логику. Обычно он состоит из компонентов, некоторые из которых предоставляют интерфейсы сервисов, доступные для использования другими участниками взаимодействия.

– Слой доступа к данным.

Этот слой обеспечивает доступ к данным, хранящимся в рамках системы, и данным, предоставляемым другими сетевыми системами. Доступ может

осуществляться через сервисы. Слой данных предоставляет универсальные интерфейсы, которые могут использоваться компонентами бизнес-слоя.

Сервисы и слои

В первом приближении решение, основанное на сервисах, можно рассматривать как набор сервисов, взаимодействующих друг с другом путем передачи сообщений. Концептуально эти сервисы можно считать компонентами решения в целом. Однако каждый сервис образован программными компонентами, как любое другое приложение, и эти компоненты могут быть логически сгруппированы в слой представления, бизнес-слой и слой данных. Другие приложения могут использовать сервисы, не задумываясь о способе их реализации. Принципы многослойного дизайна, обсуждаемые в предыдущем разделе, в равной степени применяются и к основанным на сервисах решениям.

Слой сервисов

Обычным подходом при создании приложения, которое должно обеспечивать сервисы для других приложений, а также реализовывать непосредственную поддержку клиентов, является использование слоя сервисов, который предоставляет доступ к бизнес-функциональности приложения (рис. 2.2).

Слой сервисов обеспечивает альтернативное представление, позволяющее клиентам использовать другой механизм для доступа к приложению.

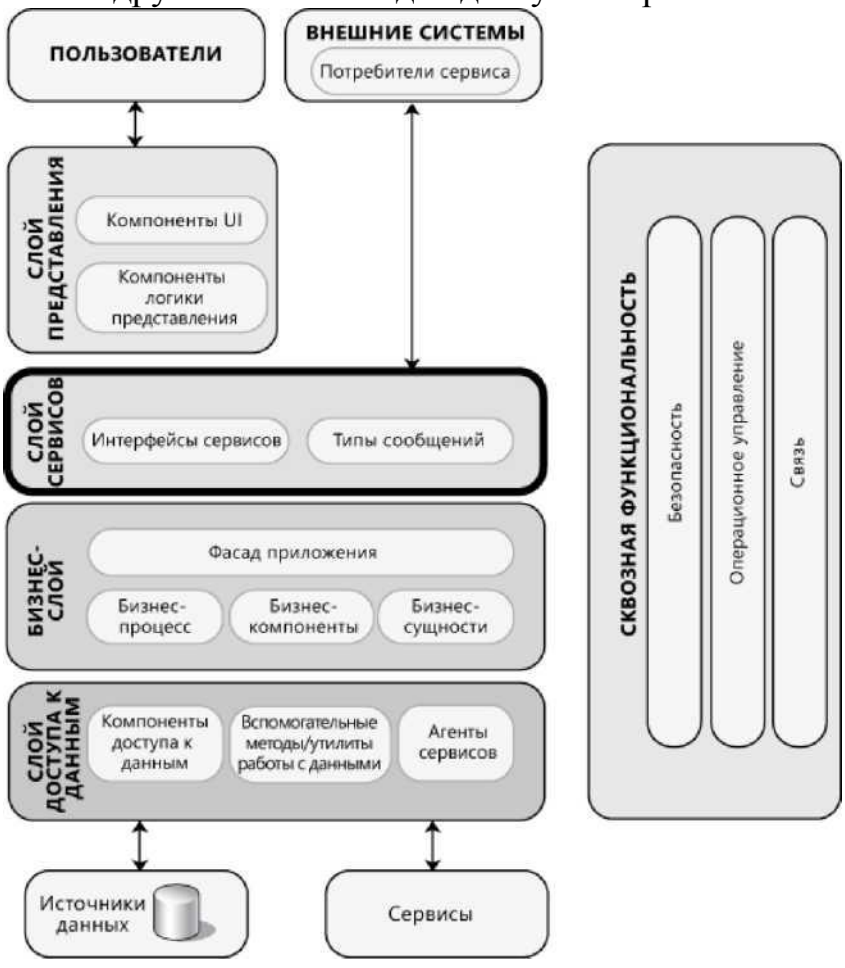

Рис. 2.2 Включение слоя сервисов в приложение

В данном сценарии пользователи могут выполнять доступ к приложению через слой представления, который обменивается данными с компонентами бизнес-слоя либо напрямую, либо через фасад приложения в бизнес-слое, если методы связи требуют композиции функциональности. Между тем, внешние клиенты и другие системы могут выполнять доступ к приложению и использовать его функциональность путем взаимодействия с бизнес-слоем через интерфейсы сервисов. Это улучшает возможности приложения для поддержки множества типов клиентов, способствует повторному использованию и более высокому уровню композиции функциональности в приложениях.

В некоторых случаях слой представления может взаимодействовать с бизнес-слоем через слой сервисов. Но это не является обязательным условием. Если физически слой представления и бизнес-слой располагаются на одном уровне, они могут взаимодействовать напрямую.

2.2 Этапы проектирования многослойной структуры

Приступая к проектированию приложения, прежде всего, сосредоточьтесь на самом высоком уровне абстракции и начинайте с группировки функциональности в слои.

Далее следует определить открытый интерфейс для каждого слоя, который зависит от типа создаваемого приложения.

Определив слои и интерфейсы, необходимо принять решение о том, как будет развертываться приложение. Наконец, выбираются протоколы связи для обеспечения взаимодействия между слоями и уровнями приложения. Несмотря на то, что разрабатываемая структура и интерфейсы могут изменяться со временем, особенно в случае применения гибкой разработки, следование этим этапам гарантированно обеспечит рассмотрение всех важных аспектов в начале процесса.

Обычно при проектировании используется следующая последовательность шагов:

Шаг 1 - Выбор стратегии разделения на слои.

Шаг 2 - Выбор необходимых слоев.

Шаг 3 - Принятие решения о распределении слоев и компонентов.

Шаг 4 - Выяснение возможности сворачивания слоев.

Шаг 5 - Определение правил взаимодействия между слоями.

Шаг 6 - Определение сквозной функциональности.

Шаг 7 - Определение интерфейсов между слоями.

Шаг 8 - Выбор стратегии развертывания.

Шаг 9 - Выбор протоколов связи.

Шаг 1 - Выбор стратегии разделения на слои

Разделение на слои представляет логическое распределение компонентов приложения по группам, выполняющим определенные роли и функции.

Использование многослойного подхода может повысить удобство обслуживания приложения и упростить его масштабирование, если необходимо повысить производительность.

Существует множество разных способов группировки взаимосвязанных функций в слои. Однако неправильное разделение на слои (слишком мало или слишком много) может лишь усложнить приложение, приводя к снижению общей производительности, удобства обслуживания и гибкости. Определение соответствующего уровня детализации при разделении приложения на слои – критически важный первый шаг в определении стратегии разделения на слои.

Также следует учесть, применяются ли слои исключительно для логического разделения либо для обеспечения физического разделения в случае необходимости. Пересечение границ слоев, особенно границ между физически удаленными компонентами, обусловливает возникновение издержек и снижение производительности. Однако общее повышение масштабируемости и гибкости приложения может быть намного более сильным аргументом по сравнению с падением производительности. Кроме того, разделение на слои может упростить оптимизацию производительности отдельных слоев без влияния на смежные слои.

В случае логического разделения взаимодействующие слои приложения будут развертываться на одном уровне и выполняться в одном процессе, что позволит применять более производительные механизмы связи, такие как прямые вызовы через интерфейсы компонентов. Однако чтобы воспользоваться преимуществами логического разделения на слои и гарантировать гибкость в будущем, следует серьезно и тщательно подойти к вопросам обеспечения инкапсуляции и слабого связывания между слоями.

Для слоев, развертываемых на разных уровнях (разные физические компьютеры), взаимодействие со смежными слоями будет происходить по сети, и необходимо обеспечить, чтобы выбранный дизайн поддерживал подходящий механизм связи, который будет учитывать задержку связи и обеспечивать слабое связывание между слоями.

Определение того, какие слои приложения вероятнее всего будут развертываться на разных уровнях, а какие на одном, также является важной часть стратегии разделения на слои. Для обеспечения гибкости необходимо гарантированно обеспечить слабую связанность слоев. Это позволяет использовать преимущества большей производительности при размещении слоев на одном уровне и в случае необходимости развертывать их на множестве уровней.

Применение многослойного подхода может несколько усложнить дизайн и увеличить продолжительность подготовительного этапа разработки, но в случае правильной реализации существенно улучшит обслуживаемость, расширяемость и гибкость приложения.

Сопоставьте преимущества, обеспечиваемые возможностью повторного использования и слабым связыванием при разделении на слои, с негативными последствиями их применения, такими как снижение производительности и повышение сложности. Тщательно продумайте, как разделить приложение на слои, и как слои будут взаимодействовать друг с другом; тем самым вы обеспечите хороший баланс производительности и гибкости. Как правило, выигрыш в гибкости и удобстве обслуживания, обеспечиваемый многослойной

схемой, намного превышает сомнительное повышение производительности, которого можно достичь в тесно связанном дизайне, не использующем слои.

Шаг 2 - Выбор необходимых слоев

Существует множество разных способов группировки взаимосвязанных функций в слои.

Самый распространенный в бизнес-приложениях подход – распределение функциональности представления, сервисов, доступа к данным и бизнес-функциональности по разным слоям. В некоторых приложениях также используются слои составления отчетов, управления и инфраструктуры.

Внимательно подходите к вопросу введения дополнительных слоев. Слой должен обеспечивать логическую группировку взаимосвязанных компонентов, которая заметно увеличивает удобство обслуживания, масштабируемость и гибкость приложения. Например, если приложение не предоставляет сервисы, возможно, отдельный слой сервисов не понадобится, тогда приложение будет включать только слой представления, бизнес-слой и слой доступа к данным.

Шаг 3 - Принятие решения о распределении слоев и компонентов

Слои и компоненты должны распределяться по разным физическим уровням, только если в этом есть необходимость.

К типовым причинам реализации распределенного развертывания относятся политики безопасности, физические ограничения, совместно используемая бизнес-логика и масштабируемость.

Если компоненты представления Веб-приложения осуществляют синхронный доступ к компонентам бизнес-слоя и ограничения безопасности не требуют наличия границы доверия между слоями, рассмотрите возможность развертывания компонентов бизнес-слоя и слоя представления на одном уровне, это обеспечит максимальную производительность и управляемость.

В насыщенных клиентских приложениях, в которых обработка UI выполняется на клиентском компьютере, вариант развертывания компонентов бизнес-слоя на отдельном бизнес-уровне может быть выбран по соображениями безопасности и для повышения управляемости.

Развертывайте бизнес-сущности на одном уровне с кодом, их использующим. Это может означать их развертывание в нескольких местах, например, размещение копий на отдельном уровне представления или данных, логика которого использует или ссылается на эти бизнес-сущности. Развертывайте компоненты агентов сервиса на том же уровне, что и код, вызывающий эти компоненты, если ограничения безопасности не требуют наличия границы доверия между ними.

Рассмотрите возможность развертывания асинхронных компонентов бизнес-слоя, компонентов рабочего процесса и сервисов с одинаковыми характеристиками загрузки и ввода/вывода на отдельном уровне. Это позволит настраивать инфраструктуру для обеспечения максимальной производительности и масштабируемости.

Шаг 4 - Выяснение возможности сворачивания слоев

В некоторых случаях имеет смысл свернуть слои. Например, в приложении, имеющем очень ограниченный набор бизнес-правил или использующем правила преимущественно для валидации, бизнес-логика и логика представления могут быть реализованы в одном слое. В приложении, которое просто извлекает данные с Веб-сервиса и отображает их, может иметь смысл просто добавить ссылки на Веб-сервис непосредственно в слой представления и использовать данные Веб-сервиса напрямую. В этом случае логически объединяются слои доступа к данным и представления.

Это лишь некоторые примеры того, когда имеет смысл сворачивание слоев. Тем не менее, группировка функциональности в слои является общим правилом. В некоторых случаях слой может выступать в роли прокси- или транзитного уровня, который обеспечивает инкапсуляцию или слабое связывание практически без предоставления функциональности. Но отделение этой функциональности позволит расширять ее в будущем без оказания влияния или с небольшим влиянием на другие слои.

Шаг 5 - Определение правил взаимодействия между слоями

Когда дело доходит до стратегии разделения на слои, необходимо определить правила взаимодействия слоев друг с другом.

Основная цель задания правил взаимодействия – минимизация зависимостей и исключение циклических ссылок. Например, если два слоя имеют зависимости от компонентов третьего слоя, появляется циклическая зависимость.

Общим правилом, которого следует придерживаться в данном случае, является разрешение только однонаправленного взаимодействия между слоями через применение одного из следующих подходов:

- Взаимодействие сверху вниз.

Слои могут взаимодействовать со слоями, расположенными ниже, но нижние слои никогда не могут взаимодействовать с расположенными выше слоями. Это правило поможет избежать циклических зависимостей между слоями. Использование событий позволит оповещать компоненты расположенных выше слоев об изменениях в нижних слоях без введения зависимостей.

- Строгое взаимодействие.

Каждый слой должен взаимодействовать только со слоем, расположенным непосредственно под ним. Это правило обеспечит строгое разделение, при котором каждый слой знает только о слое сразу под ним. Положительный эффект от этого правила в том, что изменения в интерфейсе слоя будут оказывать влияние только на слой, расположенный непосредственно над ним. Применяйте данный подход при проектировании приложения, которое предполагается расширять новой функциональностью в будущем, если хотите максимально сократить воздействие этих изменений; или при проектировании приложения, для которого необходимо обеспечить возможность распределения на разные уровни.

– Свободное взаимодействие.

Более высокие слои могут взаимодействовать с расположенными ниже слоями напрямую, в обход других слоев. Это может повысить производительность, но также увеличит зависимости. Иначе говоря, изменения в нижнем слое может оказывать влияние на несколько расположенных выше слоев. Этот подход рекомендуется применять при проектировании приложения, которое гарантированно будет размещаться на одном уровне (например, самодостаточное насыщенное клиентское приложение), или при проектировании небольшого приложения, для которого внесение изменений, затрагивающих множество слоев, не потребует больших усилий.

Шаг 6 - Определение сквозной функциональности

Определившись со слоями, необходимо обратить внимание на функциональность, охватывающую все слои. Такую функциональность часто называют сквозной функциональностью. К ней относится протоколирование, валидация, аутентификация и управление исключениями. Важно выявить все сквозные функции приложения и по возможности спроектировать для каждой из них отдельные компоненты. Такой подход поможет обеспечить лучшую возможность повторного использования и обслуживания.

Избегайте смешения такого общего кода с кодом компонентов слоев, чтобы слои и их компоненты вызывали компоненты сквозной функциональности только для выполнения таких действий, как протоколирование, кэширование или аутентификация. Поскольку эта функциональность должна быть доступна для всех слоев, способ развертывания компонентов сквозной функциональности должен обеспечивать это, даже если слои физически размещаются на разных уровнях.

Существуют разные подходы к реализации сквозной функциональности, от общих библиотек, таких как Enterprise Library группы patterns & practices, до методов аспектно-ориентированное программирование (Aspect Oriented Programming (AOP)), в которых код сквозной функциональности вставляется прямо в откомпилированный файл с помощью метаданных.

Шаг 7 - Определение интерфейсов между слоями

Основная цель при определении интерфейса слоя – обеспечить слабое связывание между слоями. Это означает, что слой не должен раскрывать внутренние детали, от которых может зависеть другой слой. Вместо этого интерфейс слоя должен быть спроектирован так, чтобы свести до минимума зависимости путем предоставления открытого интерфейса, скрывающего детали компонентов слоя. Такое сокрытие называется абстракцией.

Существует множество способов реализовать ее. Следующие подходы могут использоваться для определения интерфейса слоя:

– Абстрактный интерфейс.

Может быть определен с помощью абстрактного базового класса или интерфейса, который выступает в роли описания типа для конкретных классов. Этот тип определяет общий интерфейс, используемый для взаимодействия с

этим слоем. Такой подход также улучшает тестируемость, потому что позволяет использовать тестовые объекты (иногда называемые mock-объектами или фиктивными объектами), реализующие абстрактный интерфейс.

– Общий тип проектирования.

Многие шаблоны проектирования определяют конкретные типы объектов, которые представляют интерфейс в разных слоях. Эти типы объектов обеспечивают абстракцию, которая скрывает детали, касающиеся слоя. Например, шаблон Table Data Gateway определяет типы объектов, которые представляют таблицы в базе данных и отвечают за реализацию SQL-запросов, необходимых для доступа к данным. Сущности, работающие с объектом, ничего не знают о SQL-запросах или деталях того, как объект подключается к базе данных и выполняет команды. Многие шаблоны проектирования базируются на абстрактных интерфейсах, но в основе некоторых из них лежат конкретные классы. Большинство шаблонов, такие как Table Data Gateway, хорошо задокументированы в этом отношении. Общие типы проектирования следует применять, если необходим способ быстро и просто реализовать интерфейс слоя или при реализации шаблона проектирования для интерфейса слоя.

– Инверсия зависимостей.

Это такой стиль программирования, при котором абстрактные интерфейсы определяются вне или независимо от слоев. Тогда слои зависят не друг от друга, а от общих интерфейсов. Шаблон Dependency Injection является типовой реализацией инверсии зависимостей. При использовании Dependency Injection контейнер описывает сопоставления, определяющие как находить компоненты, от которых могут зависеть другие компоненты, и контейнер может создавать и вводить эти зависимые компоненты автоматически. Подход с инверсией зависимостей обеспечивает гибкость и может помочь в реализации модульного дизайна, поскольку зависимости определяются конфигурацией, а не кодом. Также такой подход максимально упрощает тестирование, потому что позволяет вводить тестовые классы на разные уровни дизайна.

– Основанный на обмене сообщениями.

Вместо взаимодействия с компонентами других слоев напрямую через вызов их методов или доступ к свойствам можно использовать связь посредством обмена сообщениями для реализации интерфейсов и обеспечения взаимодействия между слоями. Существует несколько решений для обмена сообщениями, такие как Windows Communication Foundation, Веб-сервисы и Microsoft Message Queuing, которые поддерживают взаимодействие через физические границы и границы процессов. Можно также комбинировать абстрактные интерфейсы с общим типом сообщений, используемым для определения структур данных для взаимодействия. Основное отличие подхода на основе сообщений в том, что для взаимодействия между слоями используется общий интерфейс, инкапсулирующий все детали взаимодействия. Этот интерфейс может определять операции, схемы данных, контракты уведомления о сбоях, политики системы безопасности и многие другие аспекты, относящиеся к обмену данными между слоями. Основанный на обмене сообщениями подход рекомендуется применять при реализации Веб-приложения и описании

интерфейса между слоем представления и бизнес-слоем, который должен поддерживать множество типов клиентов, или если требуется поддерживать взаимодействие через физические границы и границы процессов. Также рассмотрите возможность применения такого подхода, если хотите формализовать взаимодействие или взаимодействовать с интерфейсом, не сохраняющим состояние, когда данные о состоянии передаются с сообщением.

Для реализации взаимодействия между слоем представления Веб-приложения и слоем бизнес-логики рекомендуется использовать подход на основе сообщений. Если бизнес-слой не сохраняет состояния между вызовами (другими словами, каждый вызов между слоем представления и бизнес-слоем представляет новый контекст), можно передавать данные контекста вместе с запросом и обеспечить общую модель обработки исключений и ошибок в слое представления.

Шаг 8 - Выбор стратегии развертывания

Существует несколько общих шаблонов, которые представляют структуры развертывания приложений, применяемые во многих решениях. Когда требуется выбрать наиболее подходящее решение развертывания для приложения, полезно сначала рассмотреть общие шаблоны. Только полностью разобравшись с разными схемами развертывания, можно переходить к конкретным сценариям, требованиям и ограничениям безопасности, чтобы определиться с наиболее подходящим шаблоном или шаблонами.

Шаг 9 - Выбор протоколов связи

Физические протоколы, используемые для связи между слоями или уровнями, играют основную роль в обеспечении производительности, безопасности и надежности приложения. Выбор протокола связи имеет еще большее значение для распределенного развертывания. Если компоненты размещаются физически на одном уровне, часто можно положиться на прямое взаимодействие этих компонентов. Но если компоненты и слои развернуты физически на разных серверах и клиентских компьютерах, как это происходит в большинстве сценариев, необходимо продумать, как обеспечить эффективную и надежную связь между компонентами этих слоев.

2.3 Общие принципы проектирования

Рекомендации по проектированию слоя представления

Слой представления содержит компоненты, реализующие и отображающие пользовательский интерфейс, а также управляющие взаимодействием с пользователем. Этот слой, кроме компонентов, организовывающих взаимодействие с пользователем, включает элементы управления для ввода данных пользователем и их отображения. На рис. 1.1 показано место слоя представления в общей архитектуре приложения.

Слой представления обычно включает следующие компоненты:

- Компоненты пользовательского интерфейса.

Это визуальные элементы приложения, используемые для отображения данных пользователю и приема пользовательского ввода.

– Компоненты логики представления.

Логика представления – это код приложения, определяющий поведение и структуру приложения и не зависящий от конкретной реализации пользовательского интерфейса. При реализации шаблона Separated Presentation могут использоваться следующие компоненты логики представления: Презентатор (Presenter), Модель презентации (Presentation Model) и Модель Представления (View Model). Слой представления также может включать компоненты Модели слоя представления (Presentation Layer Model), которые инкапсулируют данные бизнес-слоя, или компоненты Сущности представления (Presentation Entity), которые инкапсулируют бизнес-логику и данные в форме, удобной для использования слоем представления.

Принципы проектирования слоя представления

При проектировании слоя представления необходимо учесть несколько основных факторов. Чтобы создаваемая конструкция гарантированно отвечала требованиям приложения, следуйте лучшим практикам и руководствуйтесь такими принципами:

– Выбирайте соответствующий тип приложения.

От выбранного типа приложения будут существенно зависеть доступные варианты реализации слоя представления. Определитесь, будете ли вы реализовывать насыщенный (смарт) клиент, Веб-клиент или насыщенное Интернет-приложение (Rich Internet Application, RIA). Решение должно приниматься на основании требований, предъявляемых к приложению, и ограничений, накладываемых организацией или средой.

– Выбирайте соответствующую технологию UI.

Разные типы приложений обеспечивают разные наборы технологий для разработки слоя представления. Каждый тип технологии обладает индивидуальными преимуществами, которые определяют возможность создания соответствующего слоя представления.

– Используйте соответствующие шаблоны.

Шаблоны слоя представления предлагают проверенные решения обычных проблем, возникающих при проектировании слоя представления. Помните, что не все шаблоны применимы в равной степени ко всем архетипам, но общий шаблон Separated Presentation, в котором аспекты, касающиеся представления, отделены от базовой логики приложения, подходит для всех типов приложений. Специальные шаблоны, такие как Model-View-Controller (MVC), Model-View-Presenter (MVP) и Supervising Presenter, обычно используются в слое представления насыщенных клиентских приложений и RIA. Разновидности шаблонов MVC и MVP могут применяться в Веб-приложениях.

– Разделяйте функциональные области.

Используйте специальные компоненты UI для формирования визуального представления, отображения и взаимодействия с пользователем. В сложных случаях, или если хотите обеспечить возможность модульного тестирования,

обратите внимание на специальные компоненты логики представления для управления обработкой взаимодействия с пользователем.

Также применение специальных сущностей представления позволит представлять бизнес-логику и данные в форме, удобной для использования компонентами UI и логики представления. Сущности представления инкапсулируют бизнес-логику и данные бизнес-слоя в рамках слоя представления и используют их во многом так же, как используются бизнес-сущности в бизнес-слое.

- Учитывайте рекомендации по проектированию пользовательского интерфейса.

При проектировании слоя представления придерживайтесь рекомендаций своей организации по UI, включая такие аспекты, как удобство и простота доступа, локализация и удобство использования. Для выбранных типа клиента и технологий ознакомьтесь с установленными рекомендациями по интерактивности UI, удобству использования, совместимости с системой, соответствию предъявляемым требованиям, а также с шаблонами проектирования UI, и примените те из них, которые соответствуют дизайну и требованиям создаваемого приложения.

- Придерживайтесь принципов ориентированного на пользователя проектирования.

До того, как приступать к проектированию слоя представления, поймите своего потребителя. Используйте опросы, исследования удобства использования, и интервью для выбора варианта пользовательского интерфейса, наиболее соответствующего требованиям заказчика.

Рекомендации по проектированию бизнес-слоя

На рис. 1.1 показано место бизнес-слоя в общей архитектуре приложения. Бизнес-слой обычно включает следующие компоненты:

- Фасад приложения.

Этот необязательный компонент обычно обеспечивает упрощенный интерфейс для компонентов бизнес-логики, часто сочетая множество бизнес-операций в одну, что упрощает использование бизнес-логики. Это сокращает количество зависимостей, потому что вызывающим сторонам извне нет необходимости знать детали компонентов бизнес-слоя и отношения между ними.

- Компоненты бизнес-логики.

Бизнес-логика, как и любая логика приложения, занимается вопросами извлечения, обработки, преобразования и управления данными приложения; применением бизнес-правил и политик и обеспечением непротиворечивости и действительности данных. Чтобы создать наилучшие условия для повторного использования, компоненты бизнес-логики не должны содержать поведения или логики приложения конкретного варианта использования или пользовательской истории. Компоненты бизнес-логики можно подразделить на две категории:

  - Компоненты бизнес-процесса.

После того, как компоненты UI получили необходимые данные от пользователя и передали их в бизнес-слой, приложение может использовать эти данные для осуществления бизнес- процесса. Большинство бизнес-процессов включают множество этапов, которые должны выполняться в установленном порядке и могут взаимодействовать друг с другом через различные механизмы координирования. Компоненты бизнес-процесса определяют и координируют долгосрочные многоэтапные бизнес-процессы и могут быть реализованы с помощью инструментов управления бизнес-процессами. Они работают с компонентами бизнес-процесса, которые создают экземпляры и осуществляют операции с компонентами рабочего процесса.

- Компоненты бизнес-сущностей.

Бизнес-сущности, или более общее название - бизнес-объекты, инкапсулируют бизнес-логику и данные, необходимые для представления в приложении объектов реального мира, таких как заказчики (Customers) или заказы (Orders). Они хранят значения данных и предоставляют их через свойства; содержат бизнес-данные приложения и управляют ими; и предоставляют программный доступ с сохранением состояния к бизнес- данным и связанной функциональности. Бизнес-сущности также проверяют данные, содержащиеся в сущности; они инкапсулируют бизнес-логику для обеспечения непротиворечивости данных, а также реализации бизнес-правил и поведения.

Принципы проектирования бизнес-слоя

При проектировании бизнес-слоя перед архитектором ПО стоит задача максимально упростить приложение путем разделения задач на разные функциональные области. Например, логика обработки бизнес-правил, бизнес-процессов и бизнес-сущностей – все они представляют разные функциональные области. В рамках каждой области проектируемые компоненты должны быть сфокусированы на решении конкретной задачи и не должны включать код, связанный с другими функциональными областями.

При проектировании бизнес-слоя придерживайтесь следующих принципов:

– Убедитесь в необходимости отдельного бизнес-слоя.

По возможности всегда создавайте отдельный бизнес-слой, это способствует повышению удобства обслуживания приложения. Исключением могут быть лишь приложения с небольшим числом или вообще без бизнес-правил (кроме валидации данных).

– Определитесь с ответственностями и потребителями бизнес-слоя.

Это поможет принять решение о том, какие задачи должен выполнять бизнес-слой, и каким образом будет предоставляться доступ к нему. Используйте бизнес-слой для обработки сложных бизнес-правил, преобразования данных, применения политик и валидации. Если бизнес-слой будет использоваться и слоем представления, и внешними приложениями, можно предоставить бизнес-слой в виде сервиса.

– Не смешивайте в бизнес-слое компоненты разных типов.

Используйте бизнес-слой как средство избежать смешения кода представления и доступа к данным с бизнес-логикой, чтобы отделить

бизнес-логику от логики представления и доступа к данным и упростить тестирование бизнес-функциональности. Также используйте бизнес-слой, чтобы централизовать функции бизнес-логики и обеспечить возможность повторного использования.

– Сократите количество сетевых вызовов при доступе к удаленному бизнес-слою.

Если бизнес-слой размещается на другом уровне, физически отдельно от других слоев и клиентов, с которыми должен взаимодействовать, рассмотрите возможность реализации удаленного управляемого сообщениями фасада приложения или слоя сервисов, который объединит мелкие операции в более крупные. Используйте большие пакеты для передачи данных по сети, такие как Объекты переноса данных (Data Transfer Objects, DTO).

– Избегайте тесного связывания между слоями.

При создании интерфейса бизнес-слоя применяйте принципы абстракции для максимального ослабления связывания. К техникам абстракции относятся использование открытых объектных интерфейсов, общих описаний интерфейсов, абстрактных базовых классов или связи через обмен сообщениями. Для Веб-приложений создайте управляемый сообщениями интерфейс между слоем представления и бизнес-слоем.

Рекомендации по проектированию слоя доступа к данным

На рис. 1.1 показано место слоя доступа к данным в общей архитектуре приложения.

Слой доступа к данным может включать следующие компоненты:

– Компоненты доступа к данным.

Эти компоненты абстрагируют логику, необходимую для доступа к базовым хранилищам данных. Они обеспечивают централизацию общей функциональности доступа к данным, что способствует упрощению настройки и обслуживания приложения. Некоторые инфраструктуры доступа к данным могут требовать, чтобы общая логика доступа к данным была определена и реализована в отдельных вспомогательных или служебных компонентах доступа к данным, пригодных для повторного использования. Другие инфраструктуры доступа к данным, включая многие инфраструктуры объектно-реляционного сопоставления (Object/Relational Mapping, O/RM), реализуют такие компоненты автоматически, сокращая объем кода доступа к данным, который должен написать разработчик.

– Агенты сервисов.

Если компонент бизнес-слоя должен выполнять доступ к данным, предоставляемым внешним сервисом, может понадобиться реализовать код управления семантикой взаимодействия с этим конкретным сервисом. Агенты сервисов реализуют компоненты доступа к данным, которые изолируют меняющиеся требования вызова сервисов от приложения и могут обеспечивать дополнительные сервисы, такие как кэширование, поддержку работы в автономном режиме и базовое преобразование между форматом данных,

предоставляемых сервисом, и форматом, поддерживаемым вашим приложением.

Принципы проектирования слоя данных

Слой доступа к данным должен отвечать требованиям приложения, работать эффективно и безопасно и обеспечивать простоту обслуживания и расширения в случае изменения бизнес- требований.

При проектировании слоя доступа к данным руководствуйтесь следующими общими принципами:

- Правильно выберите технологию доступа к данным.

Выбор технологии доступа к данным зависит от типа данных, с которыми придется работать, и того, как предполагается обрабатывать данные в приложении. Для каждого сценария существуют наиболее подходящие технологии.

- Используйте абстракцию для реализации слабо связанного интерфейса слоя доступа к данным.

Этот подход можно реализовать путем определения интерфейсных компонентов, таких как шлюз с общеизвестными входными и выходными параметрами, который преобразует запросы в формат, понятный компонентам слоя. Кроме того, с помощью интерфейсных типов или абстрактных базовых классов можно определить совместно используемую абстракцию, которая должна быть реализована интерфейсными компонентами.

- Инкапсулируйте функциональность доступа к хранилищу данных в слое доступа к данным.

Слой доступа к данным должен скрывать детали доступа к источнику данных. Он должен обеспечивать управление подключениями, формирование запросов и сопоставление сущностей приложения со структурами источника данных. Потребители слоя доступа к данным взаимодействуют с ним через абстрактные интерфейсы с использованием сущностей приложения, таких как объекты предметной области, типизированные наборы данных (Typed DataSet) и XML, и не должны знать внутренних деталей слоя доступа к данным. Такое разделение функциональных областей упрощает разработку и обслуживание приложения.

- Примите решение о том, как будет выполняться сопоставление сущностей приложения со структурами источника данных.

Тип сущности, используемой в приложении, является основным фактором при принятии решения о методе сопоставления этих сущностей со структурами источника данных. Обычно для этого используются шаблоны Domain Model или Table Module либо механизмы Объектнореляционного сопоставления (Object/Relational Mapping, O/RM), однако, бизнес-сущности могут быть реализованы с использованием разнообразных подходов. Необходимо определить стратегию заполнения бизнес-сущностей или структур данных из источника данных и их предоставления для доступа с бизнес-слоя или слоя представления приложения.

- Рассмотрите возможность объединения структур данных.

При предоставлении данных через сервисы воспользуйтесь Объектами передачи данных (Data Transfer Objects, DTO), они помогут организовать данные в унифицированные структуры. Кроме того, объекты DTO позволяют реализовать слабо детализированные операции, обеспечивая при этом структуру для перемещения данных через границы. Объекты DTO также могут объединять бизнес-сущности при выполнении агрегированных операций. При использовании шаблона Table Data Gateway или Active Record данные могут быть представлены с помощью класса DataTable (Таблица данных).

- Примите решение о том, как будет реализовано управление подключениями.

Обычно слой доступа к данным должен создавать и управлять подключениями ко всем источникам данных, используемым приложением. Необходимо выбрать метод хранения и защиты данных подключения, соответствующий требованиям безопасности предприятия. Возможными вариантами могут быть шифрование разделов конфигурационного файла или сохранение конфигурационных данных исключительно на сервере.

- Определите, как будут обрабатываться исключения, возникающие при обработке данных.

Слой доступа к данным должен перехватывать и обрабатывать (по крайней мере, обеспечивать начальный этап) все исключения, связанные с источниками данных и операциями CRUD (Create, Read, Update и Delete). Исключения, касающиеся самих данных и доступа к источникам данных, и ошибки истечения времени ожидания должны обрабатываться в этом слое и передаваться в другие слои, только если сбои оказывают влияние на скорость ответа и функциональность приложения.

- Учтите риски безопасности.

Слой доступа к данным должен обеспечивать защиту от попыток похищения или повреждения данных и защищать механизмы, используемые для получения доступа к источнику данных. Например, очищать детализированную информацию об ошибках или исключениях, чтобы не было возможности разглашения данных из источника данных, и использовать менее привилегированные учетные записи, обладающими только необходимыми для осуществления операций приложения правами. Даже если сам источник данных обладает возможностью ограничивать привилегии, защита должна быть реализована и в слое доступа к данным, и в источнике данных. Доступ к базе данных должен осуществляться через параметризованные запросы, чтобы предотвратить возможность успеха атак с внедрением SQL-кода. Никогда не применяйте конкатенацию строк для построения динамических запросов на основе вводимых пользователем данных.

- Сократите количество сетевых вызовов и обращений к базе данных.

Рассмотрите возможность группировки команд в одну операцию базы данных.

- Учтите требования производительности и масштабируемости.

Для слоя доступа к данным требования масштабируемости и производительности должны учитываться во время проектирования. Например, для приложения электронной коммерции именно производительность слоя доступа к данным, скорее всего, будет «узким местом» приложения. Если производительность слоя доступа к данным критична, используйте инструменты профилирования, чтобы понять и затем сократить количество или разбить ресурсоемкие операции с данными.

Рекомендации по проектированию слоя сервисов

При предоставлении доступа к функциональности приложения через сервисы функции сервиса должны быть выделены в отдельный слой сервисов.

В слое сервисов определяется и реализуется интерфейс сервиса и контракты данных (или типы сообщений). Одним из самых важных моментов, о котором нельзя забывать, является то, что сервис никогда не должен раскрывать детали внутренних процессов или бизнес-сущностей, используемых в приложении. В частности, необходимо гарантировать, что сущности бизнес-слоя не будут оказывать большого влияния на контракты данных. Слой сервисов должен предоставлять компоненты-трансляторы, которые будут обеспечивать преобразование форматов данных между сущностями бизнес-слоя и контрактами данных.

На рис. 2.2 показано место слоя сервисов в общей архитектуре приложения.

Слой сервисов обычно включает следующие компоненты:

– Интерфейсы сервисов.

Сервисы предоставляют интерфейсы, в которые передаются все входящие сообщения. Интерфейс сервиса можно рассматривать как фасад, предоставляющий потенциальным потребителям доступ к бизнес-логике, реализованной в приложении (как правило, логику бизнес-слоя).

– Типы сообщений.

При обмене данными через слой сервисов структуры данных заключаются в структуры сообщений, поддерживающие разные типы операций. Слой сервисов также обычно включает типы и контракты данных, которые определяют используемые в сообщениях типы данных.

Принципы проектирования слоя сервисов

При проектировании слоя сервисов необходимо учесть множество факторов. Многие из этих аспектов проектирования касаются проверенных практик, имеющих отношение к созданию многослойных архитектур. Однако для сервисов необходимо рассматривать также и факторы, связанные с обменом сообщениями.

Основное, на что требуется обратить внимание: сервисы взаимодействуют посредством обмена сообщениями, как правило, по сети, что по сути своей медленнее, чем прямое взаимодействие внутри процесса, и обычно сервисы взаимодействуют с потребителями асинхронно. Кроме того, сообщения, передаваемые между сервисом и потребителем, могут быть маршрутизированы, изменены, доставлены в порядке, отличном от порядка, в котором они

отправлялись, или даже утрачены, если не реализован механизм гарантированной доставки.

Все эти аспекты требуют создания дизайна, в котором будет учтено такое недетерминированное поведение процесса обмена сообщениями. При проектировании слоя сервисов руководствуйтесь следующими принципами:

- Проектируйте сервисы, областью действия которых является приложение, а не компонент. Операции сервисов должны охватывать большие фрагменты функциональности и сосредотачиваться на операциях приложения.

Проектирование слишком узконаправленных операций может иметь негативное влияние на производительность и масштабируемость. Тем не менее, необходимо обеспечить, чтобы сервис не возвращал неограниченно большие объемы данных. Например, для сервиса, который может возвращать большие объемы демографических данных, необходимо создать операцию, которая будет возвращать не все данные за один вызов, а подмножества данных определенного размера. Размер подмножества данных должен соответствовать возможностям сервиса и требованиям его потребителей.

- Проектируйте расширяемые сервисы и контракты данных, не делая допущений о предполагаемом клиенте.

Иначе говоря, контракты данных должны быть спроектированы так, чтобы их можно было расширять, не оказывая влияния на потребителей сервиса. Однако чтобы избежать излишней сложности или для поддержки изменений, не обладающих обратной совместимостью, возможно, придется создавать новые версии интерфейса сервиса, которые будут функционировать параллельно с существующими версиями. Нельзя делать предположения о клиентах или о том, как они планируют использовать предоставляемый сервис.

- Проектируйте только соответственно контракту сервиса.

Слой сервисов должен реализовывать и обеспечивать только функциональность, оговоренную контрактом сервиса. Внутренняя реализация и детали сервиса никогда не должны раскрываться внешним потребителям. Также если возникает необходимость изменить контракт сервиса для включения новой функциональности, реализованной сервисом, и новые операции и типы не являются обратно совместимыми с существующими контрактами, рассмотрите возможность создания версий контрактов. Определите новые операции, предоставляемые сервисом в новой версии контракта сервиса, и новые типы передаваемых сообщений в новой версии контракта данных.

- Отделяйте функциональность слоя сервиса от функциональности инфраструктуры.

В слое сервисов реализация сквозной функциональность не должен сочетаться с кодом логики сервиса, поскольку это может усложнить расширение и обслуживание реализаций. Обычно сквозная функциональности реализуется в отдельных компонентах, доступных для компонентов бизнес-слоя.

- Создавайте сущности из стандартных элементов.

По возможности компонуйте сложные типы и объекты передачи данных, используемые сервисом, из стандартных элементов.

- При проектировании учитывайте возможность поступления некорректных запросов.

Никогда нельзя делать предположение о том, что все поступающие в сервис сообщения будут корректными. Реализуйте логику валидации всех входных данных на основании значений, диапазонов и типов данных, и отклоняйте недопустимые данные.

- Обеспечьте в сервисе функциональность для выявления и обработки повторяющихся сообщений (идемпотентность).

Реализация широко известных шаблонов, таких как Receiver и Replay Protection, при проектировании сервиса обеспечит, что дублирующие сообщения не будут обрабатываться, или что повторная обработка не будет влиять на результат.

- Проектируйте сервис, чтобы он мог обрабатывать сообщения, поступающие в произвольном порядке (коммутативность).

Если существует вероятность поступления сообщений в порядке, отличном от того, в котором они отправлялись, реализуйте возможность сохранения сообщений с последующей обработкой в правильном порядке.

Для каждого слоя также необходимо рассмотреть специальные вопросы проектирования, которые в руководстве [6] сгруппированы по определенным областям дизайна (кэширование, сетевое взаимодействие, композиция, управление исключениями, навигация, взаимодействие с пользователем) и раскрывают принципы выбора технологии реализации определенного слоя, аспекты производительности и этапы проектирования.

2.4 Шаблоны проектирования

Шаблоны для слоя представления

Основные шаблоны проектирования для слоя представления организованы по категориям и представлены в таблице 2.1.

Рассмотрите возможности применения этих шаблонов при принятии проектных решений для каждой из категорий.

Таблица 2.1

Категория	**Шаблоны проектирования**
Кеширование	Cache Dependency (Кэш с зависимостью). Использует внешние данные для определения состояния данных, хранящихся в кэше. Page Cache (Кэш страниц). Улучшает время ответа динамических Вебстраниц, доступ к которым осуществляется довольно часто, но сами они меняются реже и потребляют большое количество ресурсов системы для воссоздания
Композиция и компоновка	Composite View (Составное представление). Сочетает отдельные представления в композитное представление.

Категория	**Шаблоны проектирования**
	Шаблон Presentation Model (Model-View-ViewModel). Разновидность шаблона Model-View-Controller (MVC), приспособленная для современных платформ разработки UI, на которых созданием представления (View) занимаются, главным образом, дизайнеры, а не обычные разработчики. Template View (Представление по шаблону). Реализует представление общего шаблона и создает представления на базе этого шаблонного представления. Transform View (Представление с преобразованием). Преобразует данные, переданные на уровень представления, в HTML для отображения в UI. Two-Step View (Двухэтапное представление). Преобразует модель данных в логическое представление без какого-либо специального форматирования и затем преобразует это логическое представление, добавляя необходимое форматирование.
Управление исключениями	Exception Shielding (Экранирование исключений). При возникновении исключения предотвращает предоставление сервисом данных о его внутренней реализации.
Навигация	Application Controller (Контроллер приложений). Единое место обработки навигации между окнами. Front Controller (Контроллер запросов). Шаблон только для Веб, консолидирующий обработку запросов путем направления всех запросов через один объект-обработчик, который можно изменять во время выполнения с помощью декораторов. Page Controller (Контроллер страниц). Принимает ввод из запроса и обрабатывает его для конкретной страницы или действия Веб-сайта. Command (Команда). Инкапсулирует обработку запроса в отдельный командный объект с обычным интерфейсом выполнения.
Взаимодействие с пользователем	Asynchronous Callback (Асинхронный обратный вызов). Выполняет длительные задачи в отдельном потоке, выполняющемся в фоновом режиме, и обеспечивает потоку функцию для обратного вызова по завершении выполнения задачи. Chain of Responsibility (Цепочка обязанностей). Предоставляет возможность обработать запрос нескольким объектам, устраняет возможность связывания отправителя запроса с его получателем

Шаблоны для бизнес-слоя

Основные шаблоны проектирования для бизнес-слоя организованы по категориям и представлены в таблице 2.2.

Рассмотрите возможности применения этих шаблонов при принятии проектных решений для каждой из категорий.

Таблица 2.2

Категория	**Шаблоны проектирования**
Компоненты бизнес-слоя	Application Fagade (Фасад приложения). Централизует и агрегирует поведение для обеспечения унифицированного слоя сервисов. Chain of Responsibility (Цепочка обязанностей). Предоставляя возможность обработать запрос нескольким объектам, устраняет возможность связывания отправителя запроса с его получателем. Command (Команда). Инкапсулирует обработку запроса в отдельный командный объект с общим интерфейсом выполнения.
Бизнес-сущности	Domain Model (Модель предметной области). Набор бизнес- объектов, представляющих сущности предметной области и отношения между ними. Entity Translator (Транслятор сущностей). Объект, преобразующий типы данных сообщения в бизнес-типы для запросов и выполняющий обратные преобразования для ответов. Table Module (Модуль таблицы). Единый компонент, реализующий бизнес-логику для всех строк таблицы или представления базы данных.
Рабочие процессы	Data-Driven Workflow (Управляемый данными рабочий процесс). Рабочий процесс, включающий задачи, последовательность выполнения которых определяется значениями данных в рабочем процессе или системе. Human Workflow (Рабочий процесс, управляемый оператором). Рабочий процесс, включающий задачи, выполняемые вручную. Sequential Workflow (Последовательный рабочий процесс). Рабочий процесс, включающий задачи, выполняющиеся в определенной последовательности, когда выполнение одной задачи запускается только после завершения предыдущей. State-Driven Workflow (Управляемый состоянием рабочий процесс). Рабочий процесс, включающий задачи, последовательность выполнения которых определяется состоянием системы.

Шаблоны для слоя доступа к данным

Основные шаблоны организованы по категориям и представлены в таблице 2.3. Рассмотрите возможности применения этих шаблонов при принятии проектных решений для каждой из категорий.

Таблица 2.3

Категория	**Шаблоны проектирования**
Общие	Active Record (Активная запись). Включает объект доступа к данным в сущность предметной области. Data Mapper (Преобразователь данных). Реализует слой преобразования между объектами и структурой базы данных, используемый для перемещения данных из одной структуры в другую, обеспечивая при этом их независимость. Data Transfer Object (Объект передачи данных). Объект, в котором сохраняются данные, передаваемые между процессами, что обеспечивает сокращение необходимого числа вызовов методов. Domain Model (Модель предметной области). Набор бизнес-объектов, представляющих сущности предметной области и отношения между ними. Query Object (Объект запроса). Объект, представляющий запрос к базе данных. Repository (Хранилище). Представление источника данных в памяти, работающее с сущностями предметной области. Row Data Gateway (Шлюз записи данных). Объект, выступающий в роли шлюза к отдельной записи источника данных. Table Data Gateway (Шлюз таблицы данных). Объект, выступающий в роли шлюза к таблице или представлению источника данных и выполняющий сериализацию всех запросов на выбор, вставку, обновление и удаление. Table Module (Модуль таблицы). Единый компонент, реализующий бизнес-логику для всех строк таблицы или представления базы данных
Пакетная обработка	Parallel Processing (Параллельная обработка). Позволяет обрабатывать множество пакетных операций одновременно, чтобы сократить время обработки. Partitioning (Секционирование). Разбивает большие пакеты, чтобы обрабатывать их параллельно.
Транзакции	Capture Transaction Details (Перехват данных транзакции). Создает объекты базы данных, такие как триггеры и теневые таблицы, для записи всех изменений, вносимых в ходе транзакции. Coarse Grained Lock (Блокировка крупных

Категория	Шаблоны проектирования
	фрагментов данных). Одной блокировкой блокирует набор взаимосвязанных объектов. Implicit Lock (Неявная блокировка). Использует код инфраструктуры для запроса блокировки от имени кода, выполняющего доступ к совместно используемым ресурсам. Optimistic Offline Lock (Оптимистическая блокировка в автономном режиме). Обеспечивает, чтобы изменения, вносимые в одном сеансе, не конфликтовали с изменениями другого сеанса. Pessimistic Offline Lock (Пессимистическая блокировка в автономном режиме). Предотвращает конфликты, вынуждая транзакцию блокировать данные перед их использованием. Transaction Script (Сценарий транзакции). Организует бизнес-логику каждой транзакции в одну процедуру, обращаясь к базе данных напрямую либо через тонкую оболочку над базой данных

Шаблоны для слоя сервисов

Основные шаблоны организованы по категориям и представлены в таблице 2.4. Рассмотрите возможности применения этих шаблонов при принятии проектных решений для каждой из категорий.

Таблица 2.4

Категория	Шаблоны проектирования
Сетевое взаимодействие	Duplex (Двусторонний обмен сообщениями). Двунаправленный обмен сообщениями, при котором и сервис, и клиент отправляют сообщения друг другу независимо и без учета того, какой шаблон используется, OneWay (Однонаправленный) или Request-Reply (Запрос-ответ). Fire and Forget (Отправил и забыл). Однонаправленный обмен сообщениями, используемый, когда ожидать ответа нет необходимости. Reliable Sessions (Надежные сеансы). Надежная передача сообщений из конца в конец между источником и точкой назначения, не зависящая от количества или типа посредников между конечными точками. Request Response (Запрос-ответ). Механизм двунаправленного обмена сообщениями, при котором клиент ожидает ответа на каждое отправленное сообщение
Каналы обмена сообщениями	Channel Adapter (Адаптер канала). Компонент, который может выполнять доступ к API или данным приложения и на основании этих данных публиковать сообщения в канале, а

Категория	Шаблоны проектирования
	также может принимать сообщения и вызывать функции приложения. Message Bus (Шина сообщений). Структурирует связующее промежуточное ПО между приложениями как шину связи, что позволяет приложениям взаимодействовать, используя обмен сообщениями. Messaging Bridge (Мост обмена сообщениями). Компонент, соединяющий обменивающиеся сообщениями системы и тиражирующий сообщения между этими системами. Point-to-Point Channel (Канал «точка-точка»). Передача сообщения по каналу «точка-точка» гарантирует, что получит это конкретное сообщение только один получатель. Publish-Subscribe Channel (Канал публикации-подписки). Создает механизм отправки сообщений только приложениям, заинтересованным в получении этих сообщений, без идентификации получателей.
Структура сообщения	Command Message (Сообщение с командой). Структура сообщения, используемая для поддержки команд. Document Message (Сообщение с данными документа). Структура, используемая для передачи документов или структуры данных между приложениями. Event Message (Сообщение о событии). Структура, обеспечивающая надежное асинхронное уведомление о событиях между приложениями. Request-Reply (Запрос-отклик). Запрос и отклик передаются по разным каналам.
Конечная точка сообщения	Competing Consumer (Конкурирующий потребитель). Задает несколько потребителей для одной очереди сообщений и заставляет их конкурировать за право обрабатывать сообщения, что позволяет обменивающемуся сообщениями клиенту обрабатывать множество сообщений одновременно. Durable Subscriber (Постоянный подписчик). Чтобы обеспечить гарантированную доставку в сценарии без подключения, сообщения сохраняются и затем предоставляются для доступа клиенту при подключении к каналу сообщений. Idempotent Receiver (Идемпотентный получатель). Гарантирует, что сервис обрабатывает сообщение только один раз. Message Dispatcher (Диспетчер сообщений). Компонент, рассылающий сообщения множеству потребителей. Messaging Gateway (Шлюз обмена сообщениями). Инкапсулирует вызовы, осуществляемые посредством обмена сообщениями, в один интерфейс, чтобы отделить их

Категория	Шаблоны проектирования
	от остального кода приложения. Messaging Mapper (Преобразователь обмена сообщениями). Преобразует запросы в бизнес-объекты для входящих сообщений и выполняет обратный процесс для преобразования бизнес-объектов в ответные сообщения. Polling Consumer (Опрашивающий потребитель). Потребитель сервиса, который проверяет канал на наличие сообщений через равные промежутки времени. Selective Consumer (Избирательный потребитель). Потребитель сервиса, использующий фильтры для поучения только сообщений, соответствующих определенному критерию. Service Activator (Активатор сервиса). Сервис, принимающий асинхронные запросы для вызова операций в компонентах бизнес-слоя. Transactional Client (Транзакционный клиент). Клиент, который может реализовать транзакции при взаимодействии с сервисом.
Безопасность сообщений	Data Confidentiality (Конфиденциальность данных). Использует шифрование на уровне сообщений для защиты конфиденциальных данных в сообщении. Data Integrity (Целостность данных). Гарантированно обеспечивает защиту сообщений от повреждения или подделки при передаче. Data Origin Authentication (Аутентификация происхождения данных). Проводит проверку источника сообщения как расширенный метод обеспечения целостности данных. Exception Shielding (Экранирование исключений). При возникновении исключения предотвращает разглашение сервисом данных о его внутренней реализации. Federation (Объединение). Интегрированное представление данных, распределенных по многим сервисам и потребителям. Replay Protection (Защита от атак повторов). Обеспечивает идемпотентность сообщения, предотвращая возможность его перехвата и многократного выполнения злоумышленниками. Validation (Валидация). Проверяет содержимое и значения сообщений для обеспечения защиты сервиса от неправильно сформированного или злонамеренного содержимого.
Маршрутизация сообщения	Aggregator (Агрегатор). Фильтр, обеспечивающий сбор и сохранение взаимосвязанных сообщений, объединение этих сообщений и публикацию в выходном канале одного

Категория	Шаблоны проектирования
	агрегатного сообщения для дальнейшей обработки. Content-Based Router (Маршрутизатор на основе содержимого). Выполняет маршрутизацию каждого сообщения к соответствующему потребителю, исходя из содержимого сообщения, например, наличия определенных полей, заданных значений полей и т.д. Dynamic Router (Динамический маршрутизатор). Компонент, выполняющий динамическую маршрутизацию сообщения к потребителю на основании оценки условий/правил, заданных потребителем. Message Broker (Hub and Spoke) (Брокер сообщений (веерная структура)). Центральный компонент, взаимодействующий с множеством приложений для получения сообщений из многих источников; определяет место назначения сообщения и направляет его в соответствующий канал. Message Filter (Фильтр сообщений). На основании заданных критериев предотвращает передачу по каналу потребителю нежелательных сообщений. Process Manager (Диспетчер процесса). Компонент, обеспечивающий маршрутизацию сообщений через множество этапов рабочего процесса.
Преобразование сообщения	Canonical Data Mapper (Канонический преобразователь данных). Использует общий формат данных для осуществления преобразований между двумя несопоставимыми форматами данных. Claim Check (Проверка утверждений). Извлекает данные из постоянного хранилища по необходимости. Content Enricher (Расширитель содержимого). Дополняет сообщения недостающими данными, полученными из внешнего источника данных. Content Filter (Фильтр содержимого). Удаляет конфиденциальные данные из сообщения и максимально сокращает объем сетевого трафика, удаляя ненужные данные из сообщения. Envelope Wrapper (Оболочка конверта). Оболочка сообщений, включающая данные заголовка, используемые, например, для защиты, маршрутизации или аутентификации сообщения. Normalizer (Нормализатор). Преобразует или трансформирует данные в общий формат обмена, если организации используют разные форматы.
REST	Behavior (Поведение). Применяется к ресурсам,

Категория	Шаблоны проектирования
	выполняющим операции. Обычно такие ресурсы не содержат состояния и поддерживают только операцию POST. Container (Контейнер). Создает шаблон сущности, обеспечивая средства для динамического добавления и/или обновления вложенных ресурсов. Entity (Сущность). Ресурсы, чтение которых может быть осуществлено операцией GET, но изменение возможно только с помощью операций PUT и DELETE. Store (Хранилище). Обеспечивает возможность создания и обновления сущностей с помощью операции PUT. Transaction (Транзакция). Ресурсы, поддерживающие транзакционные операции.
Интерфейс сервиса	Fagade (Фасад). Реализует унифицированный интерфейс для набора операций, чтобы обеспечить упрощенный интерфейс и уменьшить связанность систем. Remote Fagade (Удаленный фасад). Создает обобщенный унифицированный интерфейс для набора операций или процессов в удаленной подсистеме, обеспечивая обобщенный интерфейс для детализированных операций, чтобы упростить использование этой подсистемы и свести до минимума вызовы по сети. Service Interface (Интерфейс сервиса). Программный интерфейс, который может использоваться другими системами для взаимодействия с сервисом.
SOAP	Data Contract (Контракт данных). Схема, определяющая структуры данных, передаваемые с запросом к сервису. Fault Contracts (Контракты сбоев). Схема, определяющая ошибки или сбои, которые могут быть возвращены из запроса к сервису. Service Contract (Контракт сервиса). Схема, определяющая операции, которые может осуществлять сервис.

2.5 Проектирование компонентов приложения

Компоненты являются средством изоляции определенных наборов функций в элементах, которые могут распространяться и устанавливаться отдельно от другой функциональности.

В этом вопросе представлены типы компонентов и общие рекомендации по их созданию, обычно применяемые в слоях приложений, проектируемых с использованием многослойного подхода.

Общие рекомендации проектирования компонентов приложений:

- Применяйте принципы SOLID при проектировании классов, входящих в компонент:

    - Принцип единственности ответственности (Single responsibility). Класс должен отвечать только за один аспект.
    - Принцип открытости/закрытости (Open/closed principle). Классы должны быть расширяемыми без необходимости доработки.
    - Принцип замещения Лискова (Liskov substitution principle). Подтипы и базовые типы должны быть взаимозаменяемы.
    - Принцип отделения интерфейса (Interface segregation principle). Интерфейсы классов должны быть клиент-специфическими и узконаправленными. Классы должны предоставлять разные интерфейсы для клиентов, имеющих разные требования к интерфейсам.
    - Принцип инверсии зависимостей (Dependency inversion principle). Зависимости между классами должны заменяться абстракциями, что обеспечит возможность проектирования сверху вниз без необходимости проектирования сначала модулей нижнего уровня. Абстракции не должны зависеть от деталей - детали должны зависеть от абстракций.
- Проектируйте сильно связные компоненты. Не перегружайте компоненты введением в них невзаимосвязанной или смешанной функциональности. Например, всегда избегайте смешения в компонентах бизнес-слоя логики доступа к данным и бизнес-логики. Обеспечив связность функциональности, можно создавать сборки, включающие более одного компонента, и устанавливать компоненты в соответствующих слоях приложения, даже если эти слои разделены физически.
- Компонент не должен зависеть от внутренних деталей других компонентов. Каждый компонент или объект должен вызывать метод другого объекта или компонента, и этот метод должен знать, как обрабатывать запрос и, если необходимо, как направить его к соответствующим подкомпонентам или другим компонентам. Такой подход позволяет создавать более адаптируемые и удобные в обслуживании приложения.
- Продумайте, как компоненты будут взаимодействовать друг с другом. Для этого требуется понимать, какие сценарии развертывания должно поддерживать создаваемое приложение, должно ли оно поддерживать взаимодействие через физические границы или границы процесса, либо все компоненты будут выполняться в одном процессе.
- Не смешивайте код сквозной функциональности и прикладную логику приложения. Код, реализующий сквозную функциональность - это код, связанный с безопасностью, связью или управлением, таким как протоколированием и инструментированием. Смешение кода, реализующего эти функции, с логикой компонентов может привести к созданию плохо расширяемого и сложного в обслуживании дизайна.

- Применяйте основные принципы компонентного архитектурного стиля. Эти принципы состоят в том, что компоненты должны быть пригодными для повторного использования, заменяемыми, расширяемыми, инкапсулированными, независимыми и не зависеть от контекста.

Распределение компонентов по слоям

Каждый слой приложения содержит наборы компонентов, реализующих функциональность данного слоя. Эти компоненты должны быть связными и слабо связанными, чтобы обеспечить возможность повторного использования и упростить обслуживание

Компоненты слоя представления

Компоненты слоя представления реализуют функциональность, необходимую для обеспечения взаимодействия пользователей с приложением. Обычно в слое представления располагаются следующие типы компонентов:

- Компоненты пользовательского интерфейса.

Конкретная реализация пользовательского интерфейса приложения инкапсулирована в компоненты пользовательского интерфейса (UI). Это визуальные элементы приложения, используемые для отображения данных пользователю и приема пользовательского ввода. Компоненты UI, спроектированные для реализации шаблона Separated Presentation, иногда называют Представлениями (Views). В большинстве случаев их роль заключается в предоставлении пользователю интерфейса, который обеспечивает наиболее соответствующее представление данных и логики приложения, а также в интерпретации пользовательского ввода и передаче его в компоненты логики представления, которые определяют влияние ввода на данные и состояние приложения. В некоторых случаях в компонентах пользовательского интерфейса может содержаться специальная логика реализации пользовательского интерфейса, однако, как правило, они включают минимальный объем логики приложения, поскольку это может негативно сказаться на удобстве обслуживания и возможности повторного использования, а также усложнить модульное тестирование.

- Компоненты логики представления.

Логика представления - это код приложения, определяющий поведение и структуру приложения таким образом, что они не зависят от какой-либо конкретной реализации пользовательского интерфейса. Компоненты логики представления, главным образом, обеспечивают реализацию вариантов использования приложения (или пользовательских историй) и координируют взаимодействия пользователя с базовой логикой и состоянием приложения независимо от UI. Также они отвечают за организацию поступающих с бизнес-слоя данных в формат, пригодный для потребления компонентами UI. Например, они могут агрегировать данные из многих источников и преобразовывать их для большего удобства отображения. Компоненты логики представления можно подразделить на две категории:

  - Компоненты Presenter, Controller, Presentation Model и ViewModel. Данные типы компонентов используются при

реализации шаблона Separated Presentation и часто инкапсулируют логику представления слоя представления. Чтобы обеспечить максимальные возможности повторного использования и удобство тестирования, эти компоненты не привязаны ни к одному конкретному классу, элементу или элементу управления UI.

- Компоненты сущностей представления. Эти компоненты инкапсулируют бизнес-логику и данные и упрощают их потребление пользовательским интерфейсом и компонентами логики представления, например, путем преобразования типов данных или агрегации данных из нескольких источников. В некоторых случаях, это бизнес-сущности бизнес-слоя, используемые напрямую слоем представления. В других случаях, они могут представлять подмножество компонентов бизнес-сущностей и создаваться специально для поддержки слоя представления приложения. Сущности представления помогают обеспечить непротиворечивость и действительность данных в слое представления. В некоторых шаблонах раздельного представления эти компоненты называют моделями.

Компоненты слоя сервисов

Приложение может предоставлять слой сервисов для взаимодействия с клиентами или использования другими системами. Компоненты слоя сервисов обеспечивают другим клиентам и приложениям способ доступа к бизнес-логике приложения и используют функциональность приложения путем обмена сообщениями по каналу связи. Обычно в слое сервисов располагаются следующие типы компонентов:

- Интерфейсы сервисов.

Сервисы предоставляют интерфейс сервисов, в который передаются все входящие сообщения. Описание набора сообщений, которыми необходимо обмениваться с сервисом для осуществления им определенной бизнес-задачи, называется контрактом. Интерфейс сервиса можно рассматривать как фасад, предоставляющий потенциальным потребителям бизнес-логику, реализованную в приложении (как правило, это логика бизнес-слоя).

- Типы сообщений.

При обмене данными в слое сервисов структуры данных заключены в структуры сообщений, поддерживающие разные типы операций. Например, существуют такие типы сообщений, как Command (Команда), Document (Документ) и другие. Типы сообщений - это контракты сообщений, используемых для взаимодействия потребителей и провайдеров сервиса. Также слой сервисов обычно предоставляет типы данных и контракты, которые определяют типы данных, используемые в сообщениях, и изолируют внутренние типы данных от данных, содержащихся в типе сообщения. Это предотвращает раскрытие внутренних типов данных внешним потребителям, что могло бы привести к сложностям с контролем версий интерфейса.

Компоненты бизнес-слоя

Компоненты бизнес-слоя реализуют основную функциональность системы и инкапсулируют соответствующую бизнес-логику. Бизнес-слой обычно включает следующие типы компонентов:

- Фасад приложения.

Этот необязательный компонент обычно обеспечивает упрощенный интерфейс для компонентов бизнес-логики зачастую путем объединения множества бизнес-операций в одну, что упрощает использование бизнес-логики и сокращает количество зависимостей, поскольку внешним вызывающим сторонам нет необходимости знать детали бизнес-компонентов и отношения между ними.

- Компоненты бизнес-логики.

Бизнес-логика - это логика приложения, связанная с извлечением, обработкой, преобразованием и управлением данными приложения; применением бизнес-правил и политик и обеспечением непротиворечивости и действительности данных. Чтобы обеспечить наилучшие условия для повторного использования, компоненты бизнес-логики не должны включать поведение или логику приложения, относящиеся к конкретному варианту использования или пользовательской истории. Компоненты бизнес-логики можно разделить на следующие две категории:

- Компоненты рабочего процесса. После того как данные введены в компоненты UI и переданы в бизнес-слой, приложение может использовать их для выполнения бизнес-процесса. Многие бизнес-процессы состоят из множества этапов, которые должны осуществляться в соответствующем порядке и могут взаимодействовать друг с другом посредством механизмов координирования. Компоненты рабочего-процесса определяют и управляют длительными многоэтапными бизнес-процессами и могут быть реализованы с использованием инструментов управления бизнес-процессами. Компоненты рабочего процесса работают с компонентами бизнес-процесса, которые создают экземпляры компонентов рабочего процесса и осуществляют операции с ними.
- Компоненты бизнес-сущностей. Бизнес-сущности, или, более обобщенно, бизнес-объекты, инкапсулируют бизнес-логику и данные, необходимые для представления в приложении элементов реального мира, таких как заказчики (Customers) или заказы (Orders). Они сохраняют значения данных и предоставляют их через свойства; содержат и управляют бизнес-данными, которые используются приложением; и обеспечивают программный доступ с сохранением состояния к бизнес-данным и соответствующей функциональности. Также бизнес-сущности проводят проверку содержащихся в них данных и инкапсулируют бизнес-логику для обеспечения

непротиворечивости данных и реализации бизнес-правил и поведения.

Очень часто бизнес-сущности должны быть доступными компонентам и сервисам как бизнес- слоя, так и слоя данных. Например, бизнес-сущности могут сопоставляться с источником данных, и к ним могут выполнять доступ бизнес-компоненты. Если слои располагаются на одном уровне, бизнес-сущности могут использоваться совместно непосредственно через указатели. Однако при этом все равно должно быть обеспечено разделение бизнес-логики и логики доступа к данным. Этого можно достичь путем перемещения бизнес-сущностей в отдельную сборку, доступную для использования сборками и бизнес-сервисов, и сервисов данных. Этот подход аналогичен использованию шаблона инверсии зависимостей, когда бизнес-сущности отделяются от бизнес-слоя и слоя данных, и их зависимость от бизнес- сущностей реализуется как совместно используемый контракт.

Компоненты слоя доступа к данным

Компоненты слоя доступа к данным обеспечивают доступ к данным, размещенным в рамках системы, и к данным, предоставляемым другими сетевыми системами. Обычно слой доступа к данным включает следующие типы компонентов:

– Компоненты доступа к данным.

Эти компоненты абстрагируют логику, необходимую для доступа к базовым хранилищам данных. Для большинства задач доступа к данным необходима общая логика, которая может быть выделена и реализована в отдельных вспомогательных компонентах, доступных для повторного использования, или подходящей вспомогательной инфраструктуре. Это может упростить компоненты доступа к данным и централизовать логику, что облегчает обслуживание. Остальные задачи, общие для компонентов слоя данных и не относящиеся ни к одному набору компонентов, могут быть реализованы как отдельные служебные компоненты. Вспомогательные и служебные компоненты часто объединяются в библиотеку или инфраструктуру, что облегчает их повторное использование в других приложениях.

– Агенты сервисов.

Если бизнес-компонент должен использовать функциональность, предоставляемую внешним сервисом, вероятно, потребуется реализовать код для управления семантикой взаимодействия с конкретным сервисом. Агенты сервисов изолируют специальные аспекты вызова разных сервисов в приложении и могут обеспечивать дополнительные сервисы, такие как кэширование, поддержка работы в автономном режиме и базовое сопоставление форматов данных, предоставляемых сервисом, и форматов, требуемым приложением.

Компоненты сквозной функциональности

Некоторые задачи необходимо выполнять во многих слоях. Компоненты сквозной функциональности реализуют специальные типы функциональности, доступ к которым могут осуществлять компоненты любого слоя. Рассмотрим основные типы компонентов сквозной функциональности:

- Компоненты для реализации безопасности. Сюда относятся компоненты, осуществляющие аутентификацию, авторизацию и валидацию.
- Компоненты для реализации задач операционного управления. Сюда относятся компоненты, реализующие политики обработки исключений, протоколирование, счетчики производительности, конфигурацию и трассировку.
- Компоненты для реализации взаимодействия. Сюда относятся компоненты, взаимодействующие с другими сервисами и приложениями.

2.6 Проектирование компонентов пользовательского интерфейса и компонентов логики представления[5]

Прежде всего, требуется понять, какие требования предъявляются к UI, и суметь выбрать соответствующую технологию. После этого уже можно принимать решения о связывании логики представления и данных с элементами управления UI. Также необходимо иметь четкое представление о требованиях к обработке ошибок и проверке в UI.

Шаг 1 - Понимание предъявляемых к UI требований

Понимание предъявляемых к UI требований - ключ к принятию решений по типу UI, технологии и типу элементов управления, используемым для его реализации. Требования к UI определяются функциональностью, которую должно поддерживать приложение, и ожиданиями пользователей.

Начните с выяснения, кто будет пользователями приложения, и понимания целей и задачей, которые эти пользователи желают реализовывать при использовании приложения.

Особое внимание уделите вопросу последовательности задач или операций; выясните, ожидают ли пользователи структурированного последовательного взаимодействия или неструктурированного взаимодействия с произвольным порядком операций, когда существует возможность выполнения множества задач одновременно. Как часть этого процесса, также выясните, какие данные понадобятся пользователям, и формат, в котором они ожидают их увидеть. Возможно, придется провести исследования, чтобы лучше понять среду, в которой пользователь будет взаимодействовать с приложением. Кроме того, рассмотрите текущие уровни взаимодействия с пользователем и сравните их с требованиями к взаимодействию с пользователем, предъявляемыми для разрабатываемого UI, чтобы убедиться в их логичности и понятности. Все эти факторы помогут создать ориентированный на пользователя дизайн.

Один из факторов, имеющих большое влияние на выбор технологии - требуемая функциональность UI. Выясните, должен ли UI предоставлять насыщенную функциональность или взаимодействие с пользователем, должен

[5] Практическое занятие № 2.

ли он обеспечивать минимальное время отклика или требует графической или анимационной поддержки. Также рассмотрите требования с точки зрения локализации к типам данных, форматам и форматам представления для таких данных, как даты, время и валюты. Кроме того, определите требования по персонализации приложения, такие как предоставление пользователю возможности менять компоновку и стили во время выполнения.

Чтобы сделать UI интуитивно понятным и простым в использовании, продумайте компоновку или композицию интерфейса, а также перемещение пользователя по UI приложения. Это поможет выбрать соответствующие элементы управления и технологии для UI. Разберитесь с тем, какие требования физического устройства отображения (такие как размер и разрешение экрана) и специальные возможности (такие как крупный текст или кнопки, рукописный ввод и т.д.) необходимо поддерживать. Примите решение о том, как будете выполнять группировку взаимосвязанных данных в разделах UI, избегать конфликтов или неоднозначностей интерфейса и выделять важные элементы. Обеспечьте пользователям возможность быстро и легко находить сведения в приложении посредством навигационных элементов управления, функций поиска, четко именованных разделов, карт сайта и других соответствующих возможностей.

Шаг 2 - Выбор необходимого типа UI

На основании предъявляемых к UI требований можно принять решение о типе UI для приложения.

Существует ряд разных типов UI, каждый из которых обладает определенными преимуществами и недостатками. Часто обнаруживается, что предъявляемым к UI требованиям соответствует несколько типов UI. Но бывают ситуации, когда ни один из типов UI не обеспечивает полностью все требования. В этом случае необходимо рассмотреть возможность создания нескольких разных типов UI, которые будут совместно использовать бизнес-логику. Примером этому может служить приложение для call-центра, некоторые из возможностей которого предоставляются клиенту для самостоятельного использования через Веб и на мобильных устройствах.

Мобильные приложения могут разрабатываться как тонкое клиентское или насыщенное клиентское приложение. Насыщенные клиентские мобильные приложения могут поддерживать сценарии без подключения или с периодическим подключением. Веб или тонкие клиентские мобильные приложения поддерживают только сценарии с подключением. Ограничением при проектировании мобильных приложений могут быть и аппаратные ресурсы.

Насыщенные клиентские приложения обычно являются автономными или сетевыми приложениями с графическим пользовательским интерфейсом, отображающим данные с помощью различных элементов управления, развертываемыми на настольном или портативном компьютере локального пользователя. Эти приложения подходят для сценариев без подключения или с периодическим подключением, поскольку выполняются на клиентском компьютере. Насыщенное клиентское приложение является хорошим выбором,

если требуется высокодинамичный UI с малым временем отклика или UI должен обеспечивать насыщенные функциональность и взаимодействие с пользователем; либо если приложение должно поддерживать как сценарии с подключением, так и сценарии без подключения, использовать ресурсы локальной системы на клиентском компьютере или интегрироваться с другими приложениями на этом компьютере.

Обычно насыщенные Интернет-приложения (RIA) – это Веб-приложения с насыщенным графическим пользовательским интерфейсом, выполняющиеся в браузере. Как правило, приложения RIA используются в сценариях с подключением. Используйте RIA, если необходимо обеспечить UI, поддерживающий динамическое взаимодействие с пользователем с малым временем отклика или использующий потоковое мультимедиа и доступный на широком диапазоне устройств и платформ. Эти приложения могут использовать вычислительные мощности клиентского компьютера, но не могут напрямую взаимодействовать с локальными ресурсами системы, такими как веб-камеры (начиная с Silverlight 4.0, RIA приложения могут работать с камерой и микрофоном), или с другими клиентскими приложениями, такими как приложения Microsoft Office.

Веб-приложения поддерживают сценарии с постоянным подключением и могут поддерживать множество разных браузеров, выполняющихся под управлением множества различных операционных систем и на разных платформах. Веб-приложение - замечательный выбор, если UI должен быть стандартизованным, доступным для широчайшего диапазона устройств и платформ и работать только при постоянном подключении к сети. Также Веб-приложения хорошо подходят, если необходимо обеспечить доступность содержимого приложения для поиска средствами поиска в Веб.

Консольные приложения предлагают альтернативный текстовый пользовательский интерфейс и обычно выполняются в командных оболочках, таких как Command window (Интерфейс командной строки) или Power Shell. Такие приложения лучше всего подходят для задач администрирования или разработки и не используются как часть многослойного дизайна приложений.

Шаг 3 - Выбор технологии UI

После того, как вы определились с типом UI для своих компонентов UI, необходимо выбрать соответствующую технологию.

Как правило, выбор зависит от выбранного типа UI. Далее рассмотрим, какие технологии подходят для каждого из типов UI:

Пользовательские интерфейсы мобильных клиентов могут быть реализованы с использованием следующих технологий создания пользовательского интерфейса:

- Microsoft .NET Compact Framework. Это версия Microsoft .NET Framework, разработанная специально для мобильных устройств. Используйте эту технологию для мобильных приложений, которые должны выполняться на устройствах без гарантированного подключения к сети.

- ASP.NET для мобильных устройств. Это версия ASP.NET, разработанная специально для мобильных устройств. ASP.NET для мобильных приложений может размещаться на сервере Internet Information Services (IIS, Информационные Интернет-службы. Используйте эту технологию для мобильных Веб-приложений, если требуется поддерживать широкий диапазон мобильных устройств и браузеров и можно рассчитывать на постоянное подключение к сети.
- Silverlight для мобильных устройств. Для этой версии Silverlight-клиента требуется, чтобы на мобильном устройстве был установлен подключаемый модуль Silverlight. Используйте эту технологию, чтобы портировать существующие Silverlight- приложения на мобильные устройства, или если желаете создавать более насыщенные UI, чем обеспечивают другие технологии.

Пользовательские интерфейсы насыщенных клиентов могут быть реализованы с использованием следующих технологий представления:

- Windows Presentation Foundation (WPF). Приложения WPF поддерживают более широкие графические возможности, такие как 2-D и 3-D графика, независимость от разрешения экрана, расширенная поддержка документов и полиграфического оформления, анимация с временной шкалой, потоковое аудио и видео и векторная графика. WPF использует расширяемый язык разметки приложений (Extensible Application Markup Language, XAML) для описания UI, связывания данных и событий. WPF также включает расширенные возможности связывания данных и описания шаблонов. WPF-приложения отделяют визуальные аспекты UI от базовой логики управления, обеспечивая тем самым поддержку совместной работы разработчика и дизайнера: разработчики могут сосредоточиться на бизнес-логике, тогда как дизайнеры занимаются внешним видом. Используйте эту технологию для создания насыщенных медиа и интерактивных пользовательских интерфейсов.
- Windows Forms. Windows Forms является частью .NET Framework с момента ее появления и идеально подходит для бизнес-приложений. Даже несмотря на существование возможностей Windows Presentation Foundation (WPF), Windows Forms будет идеальным решением для приложений, к которым не предъявляются никакие особые требования по насыщенным медиа или интерактивным возможностям, или если группа разработки уже имеет богатый опыт работы с Windows Forms.
- Windows Forms с элементами WPF. Этот подход позволяет использовать преимущества, предоставляемые элементами управления WPF, для создания более мощных UI. WPF можно добавлять в существующее приложение Windows Forms, возможно, как этап постепенного перехода к реализации полностью на WPF. Используйте этот подход для введения насыщенных медиа и интерактивных возможностей в существующие

приложения, но не забывайте, что элементы управления WPF лучше всего работают на мощных клиентских компьютерах.

- WPF с элементами Windows Forms. Этот подход позволяет дополнить WPF элементами управления Windows Forms, предоставляющими функциональность, не обеспечиваемую WPF. Добавлять элементы управления Windows Forms в UI можно с помощью элемента WindowsFormsHost из сборки WindowsFormsIntegration. Используйте этот подход, если в пользовательском интерфейсе WPF необходимы элементы управления Windows Forms, но не забывайте о некоторых ограничениях и сложностях, связанных с перекрытием элементов управления, фокусом интерфейса и методиками формирования визуального представления, используемыми разными технологиями.
- XAML Browser Application (XBAP), использующее WPF. Данная технология позволяет размещать WPF-приложение в изолированной программной среде в Microsoft Internet Explorer или Mozilla Firefox для Windows. В отличие от Silverlight, в данном случае, доступны все возможности инфраструктуры WPF лишь с некоторыми ограничениями относительно доступа к системным ресурсам из частично доверяемой изолированной программной среды. XBAP требует, чтобы на клиентском компьютере была установлена Windows Vista или .NET Framework 3.5 и подключаемый модуль браузера XBAP. Применяйте XBAP, если имеете готовое WPF-приложение, которое требуется развернуть в Веб, или если хотите использовать насыщенные возможности WPF для создания визуального представления и UI, недоступные в Silverlight.

Пользовательские интерфейсы насыщенных Интернет-приложений могут быть реализованы с использованием следующих технологий представления:

- Silverlight. Это оптимизированный для работы в браузере аналог WPF, не зависящий от платформы и браузера. По сравнению с XBAP, Silverlight меньше и быстрее устанавливается. Благодаря своему небольшому размеру и поддержке разных платформ Silverlight является хорошим выбором для графических приложений, не требующих полной поддержки графических возможностей WPF, или в случае, когда необходимо избежать установки приложения на клиенте.
- Silverlight с AJAX. Silverlight поддерживает Асинхронный JavaScript и XML (Asynchronous JavaScript and XML, AJAX) и предоставляет свою объектную модель для доступа из JavaScript, размещаемого в Веб-странице. Эту возможность можно использовать для обеспечения взаимодействия между компонентами Вебстраницы и приложением Silverlight.

Пользовательские интерфейсы Веб-приложений могут быть реализованы с использованием следующих технологий формирования представления:

- ASP.NET Web Forms. Это фундаментальная технология проектирования и реализации UI для Веб-приложений .NET. Приложение ASP.NET Web

Forms должно быть установлено только на Веб-сервере, на клиентский компьютер никакие компоненты устанавливать не надо. Используйте эту технологию для Веб-приложений, в которых не требуются дополнительные возможности, предоставляемые AJAX, Silverlight, MVC или Dynamic Data.

- ASP.NET Web Forms с AJAX. Применяйте AJAX с ASP.NET Web Forms для асинхронной обработки запросов между сервером и клиентом, что сократит время отклика, обеспечит более насыщенное взаимодействие с пользователем и сократит количество обращений к серверу. AJAX входит в состав ASP.NET в .NET Framework версии 3.5 и последующих.
- ASP.NET Web Forms с элементами управления Silverlight. Добавив элементы управления Silverlight в готовое приложение ASP.NET, его можно расширить более насыщенными возможностями создания визуального представления и взаимодействия с пользователем. И это избавит от необходимости создавать совершенно новое Silverlight-приложение. Такой подход хорош для внедрения насыщенного медиа-содержимого Silverlight в существующее Веб-приложение. Элементы управления Silverlight и включающая их Веб-страница могут взаимодействовать на стороне клиента с помощью JavaScript.
- ASP.NET MVC. Эта технология позволяет использовать ASP.NET для создания приложений на базе шаблона Model-View-Controller (MVC). Используйте данную технологию, если требуется поддерживать разработку через тестирование и обеспечить четкое разделение функциональности обработки UI и формирования визуального представления UI. Такой подход также поможет получить полный контроль над формированием HTML и избежать смешения данных представления с кодом логики обработки.
- ASP.NET Dynamic Data. Эта технология позволяет создавать управляемые данными ASP.NET-приложения, использующие модель данных Language-Integrated Query (LINQ) to Entities. Применяйте ее, если необходима модель быстрой разработки управляемых данными LOB-приложений, основанных на простом формировании шаблонов, но при этом поддерживающих полную настройку.

Шаг 4 - Проектирование компонентов представления

Следующий шаг после выбора технологии реализации UI – проектирование компонентов UI и компонентов логики представления.

Могут использоваться следующие типы компонентов представления:

- компоненты пользовательского интерфейса;
- компоненты логики представления;
- компоненты модели представления.

Эти компоненты поддерживают разделение функциональных областей в рамках слоя представления и часто используются для реализации шаблонов раздельного представления, таких как MVP (Model-View-Presenter) или MVC

(Model-View-Controller), через разделение задач обработки UI на три роли: Модель, Представление и Контроллер/Презентатор.

Такое разделение функциональности слоя представления повышает удобство обслуживания, тестируемость и возможности повторного использования. Применение абстрактных шаблонов, таких как внедрение зависимостей, также способствует упрощению тестирования логики представления.

Компоненты пользовательского интерфейса

Компоненты UI – это визуальные элементы, отображающие данные пользователю и принимающие пользовательский ввод.

В рамках отдельного слоя представления их обычно называют Представлениями (Views). При проектировании компонентов UI руководствуйтесь следующими рекомендациями:

- Разбейте страницы или окна на отдельные пользовательские элементы управления, чтобы упростить и обеспечить возможность повторного использования этих элементов управления. Выбирайте соответствующие компоненты UI и используйте преимущества возможностей связывания данных элементов управления, применяемых в UI.
- Избегайте создания иерархий наследования пользовательских элементов управления и страниц, чтобы обеспечить возможность повторного использования кода. Отдавайте предпочтение композиции, а не наследованию, и создавайте компоненты логики представления, пригодные для повторного использования.
- Создавайте специализированные элементы управления, только если это необходимо для специализированного отображения или сбора данных.

Если видите, что имеющиеся требования к UI не могут быть реализованы стандартными элементами управления, прежде чем создавать собственные специализированные элементы управления, рассмотрите возможность приобретения готового набора элементов управления.

При создании специализированных элементов управления старайтесь расширять существующие элементы управления, а не создавать элементы управления с нуля. Расширяйте существующие элементы управления путем подключения к ним поведения, а не наследования от них. Реализуйте поддержку дизайнера для специализированных элементов управления, чтобы разработчикам было проще работать с ними.

Компоненты логики представления

Компоненты логики представления занимаются не визуальными аспектами пользовательского интерфейса, к которым обычно относится проверка, ответ на действия пользователя, взаимодействие компонентов UI и координирование взаимодействий с пользователем.

Компоненты логики представления необходимы не всегда; создавайте их, только если собираетесь выполнять в слое представления большой объем обработки, которая должна быть отделена от компонентов UI, или если хотите обеспечить более благоприятные условия для модульного тестирования логики представления.

При проектировании компонентов логики представления руководствуйтесь следующими рекомендациями:

- Если UI требует сложной обработки или должен обмениваться данными с другими слоями, используйте компоненты логики представления для отделения этой обработки от компонентов UI.
- Используйте компоненты логики представления для хранения состояния, относящегося к UI, но не характерного для конкретной реализации. Старайтесь не включать в компоненты логики представления бизнес-логику или бизнес-правила, кроме валидации ввода и данных. Также избегайте реализации в компонентах логики представления логики формирования визуального представления UI или специализированной логики UI.
- С помощью компонентов логики представления обеспечьте согласованное состояние пользовательского интерфейса при восстановлении приложения после сбоя или ошибки.
- Там, где UI требует поддержки сложного рабочего процесса, создавайте отдельные компоненты рабочего процесса, использующие такую систему управления рабочим процессом, как Windows Workflow Foundation, или реализуйте собственный механизм в бизнес-слое приложения.

Компоненты модели представления

Компоненты модели представления представляют данные, поступающие с бизнес-слоя, в формате, доступном для использования UI и компонентами логики представления. Обычно модели представляют данные, и поэтому используют компоненты доступа к данным и, возможно, компоненты бизнес-слоя для сбора этих данных. Если модель также инкапсулирует бизнес-логику, ее обычно называют сущностью представления. Компоненты модели представления могут, к примеру, агрегировать данные из множества источников, преобразовывать данные для обеспечения удобства их отображения в UI, реализовывать логику проверки и помогать в представлении бизнес-логики и состояния в рамках слоя представления. Обычно они используются для реализации шаблонов раздельного представления, таких как MVP or MVC.

При проектировании компонентов модели представления руководствуйтесь следующими рекомендациями:

- Определитесь, нужны ли вам компоненты модели представления. Обычно модели слоя представления используются для отображения специальных данных или форматов слоя представления или в случае применения шаблона раздельного представления, такого как MVP или MVC.
- Работая с элементами управления с привязкой к данным, проектируйте или выбирайте соответствующие компоненты модели представления, которые можно легко связать с элементами управления UI. При использовании в качестве формата компонента модели представления специальных объектов, коллекций или наборов данных обеспечьте

реализацию ими соответствующих интерфейсов и событий для поддержки привязки данных.

- При выполнении валидации данных в слое представления размещайте код валидации в компонентах модели представления. Но также продумайте преимущества использования кода или библиотек для централизованной валидации.
- Рассмотрите требования сериализации для данных, передаваемых в компоненты модели представления, если эти данные будут передаваться по сети или сохраняться на жестком диске клиента.

Также необходимо выбрать подходящий тип данных для компонентов модели представления и сущностей представления. Этот выбор определяется требованиями, предъявляемыми к приложению, и ограничениями, налагаемыми инфраструктурой и возможностями разработки. Начните с выбора формата данных слоя представления и примите решение о том, будут ли компоненты также инкапсулировать бизнес-логику и состояние. Далее необходимо принять решение о том, как будут представляться данные в пользовательском интерфейсе.

Существуют такие общие форматы представления данных:

- Собственный класс. Используйте собственный класс, если необходимо представлять данные как сложный объект, проецируемый непосредственно на бизнес-сущности. Например, можно создать собственный объект Order, который будет представлять данные заказа. Также собственный класс может использоваться для инкапсуляции бизнес-логики и состояния и осуществления проверки в слое представления или реализации собственных свойств.
- Array (Массив) и Collection (Коллекция). Используйте массив или коллекцию, если требуется выполнить привязку данных к элементам управления, таким как окно списка или выпадающий список, в которых используются значения одного столбца.
- DataSet (Набор данных) и DataTable (Таблица данных). Используйте DataSet или DataTable при работе с простыми табличными данными и элементами управления с привязкой к данным, такими как таблица, окно списка и выпадающий список.
- Typed Dataset (Типизированный набор данных). Используйте Typed DataSet, если хотите реализовать тесное связывание с бизнес-сущностями, чтобы избежать возникновения несогласованности из-за изменений базы данных.
- XML. Этот формат полезен при работе с Веб-клиентом, когда данные могут быть встроены в Веб-страницу или извлекаться через Веб-сервис или HTTP-запрос. Выбирайте XML при работе с такими элементами управления, как дерево или таблица. Также XML легко сохранять, сериализовать и передавать по каналам связи.
- DataReader (Модуль чтения данных). Используйте DataReader в сценариях с постоянным подключением для извлечения данных в

режиме только для чтения и только для пересылки. DataReader обеспечивает эффективный способ для последовательной обработки данных, поступающих из базы данных, или для извлечения больших объемов данных. Однако он очень тесно связывает логику со схемой базы данных, что, как правило, не рекомендуется.

Сущности представления

Компоненты модели представления должны по возможности инкапсулировать и данные, поступающие с бизнес-слоя, и бизнес-логику, и поведение. Это позволяет обеспечить непротиворечивость и корректность данных в слое представления и способствует улучшению качества взаимодействия с пользователем.

В некоторых случая в роли компонентов модели представления могут выступать бизнес-сущности бизнес-слоя, используемые напрямую слоем представления. В других случая компоненты модели представления могут представлять подмножество компонентов бизнес-сущностей, в частности, разработанных для поддержки слоя представления приложения. Например, в них могут храниться данные в формате, более удобном для использования UI и компонентами логики представления. Такие компоненты иногда называют сущностями представления.

Если бизнес-слой и слой представления располагаются на клиенте, что является типовым сценарием для насыщенных клиентских приложений, бизнес-сущности обычно используются напрямую из бизнес-слоя. Однако если необходимо сохранять или обрабатывать бизнес- данные в формате или способом, отличным от формата и поведения, предоставляемыми бизнес-сущностями бизнес-слоя, можно рассмотреть возможность применения сущностей представления.

Если бизнес-слой и слой представления располагаются на разных уровнях, использование бизнес-сущностей уровнем представления может быть реализовано через их сериализацию и передачу по сети с помощью объектов передачи данных с последующим их восстановлением в виде экземпляров бизнес-сущностей на уровне представления. Или можно восстанавливать данные как сущности представления, если требуемый формат и поведения отличаются от используемых бизнес-сущностями. Этот сценарий продемонстрирован на рис. 2.3.

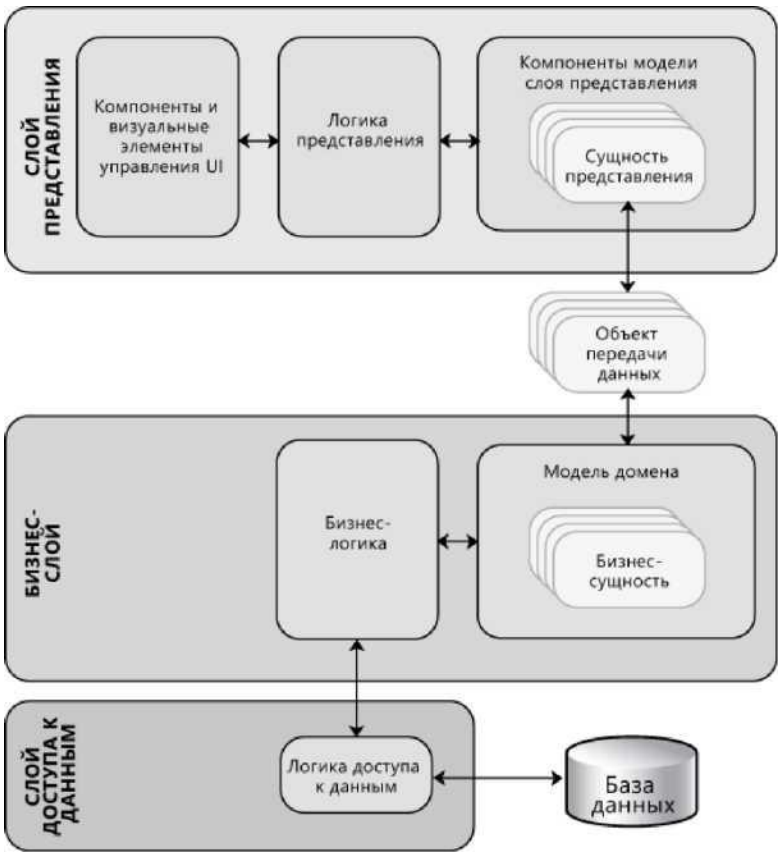

Рис. 2.3 Компоненты модели представления и сущности представления

Компоненты модели представления и сущности представления могут быть полезны при размещении слоя представления и бизнес-слоя на разных уровнях

Шаг 5 - Определение требований к привязке данных

Привязка данных обеспечивает возможность создания связи между элементами управления пользовательского интерфейса и данными или логическими компонентами приложения.

Привязка данных позволяет отображать и взаимодействовать с данными баз данных, а также данными других структур, таких как массивы и коллекции.

Привязка данных - это мост между целью привязки (обычно это элемент управления пользовательского интерфейса) и источником привязки (обычно это структура данных, модель или компонент логики представления).

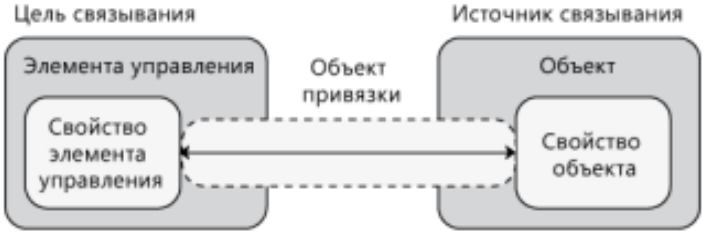

Рис.2.4 Объекты, используемые при привязке данных

Как показано на рис. 2.4, обычно в привязке данных участвуют четыре элемента, взаимодействующих друг с другом для обновления свойств связанного объекта значениями, предоставляемыми источником привязки. Элементы управления с привязкой к данным - это элементы управления, связанные с источниками данных. Например, элемент управления DataGrid

связан с коллекцией объектов. Привязка данных часто используется с шаблонами раздельного представления для связывания компонентов UI (Представлений) с презентаторами или контроллерами (компоненты логики представления) либо с моделью слоя представления или компонентами сущностей.

Поддержка привязки данных и ее реализация в каждой технологии UI разная. В общем, большинство технологий UI позволяют выполнять привязку элементов управления к объектам и спискам объектов. Но некоторые специальные технологий привязки данных могут потребовать реализации в источниках данных определенных интерфейсов и событий для обеспечения полной поддержки привязки данных, например, интерфейса INotifyPropertyChanged (Уведомление об изменении свойства) в WPF или IBindingList (Список привязки) в Windows Forms. При использовании шаблона раздельного представления логика представления и компоненты данных должны гарантированно поддерживать необходимые интерфейсы или события, чтобы обеспечить простоту привязку элементов управления UI к ним.

Обычно используются два типа привязки:

- Односторонняя привязка. Изменения свойства источника приводят к автоматическому обновлению целевого свойства, но изменения целевого свойства не распространяются на исходное свойство. Такой тип привязки подходит для неявно доступных только для чтения элементов управления. Примером такой односторонней привязки могут быть биржевые сводки. Если нет необходимости отслеживать изменения целевого свойства, использование односторонней привязки позволит избежать ненужных издержек.
- Двухсторонняя привязка. Изменения любого из свойств, исходного либо целевого, приводят к автоматическому обновлению второго свойства. Такой тип привязки подходит для редактируемых форм или других полностью интерактивных сценариев UI. Многие редактируемые элементы управления в Windows Forms, ASP.NET и WPF поддерживают двухстороннюю привязку, так что изменения источника данных отражаются в элементе управления UI, и изменения в элементе управления UI отражаются в источнике данных.

Шаг 6 - Выработка стратегии обработки ошибок

Компоненты UI являются внешней границей приложения, и поэтому должны реализовывать соответствующую стратегию обработки ошибок для обеспечения стабильности приложения и положительного впечатления при взаимодействии с пользователем. При проектировании стратегии обработки ошибок рассмотрите следующие варианты:

- Стратегия централизованной обработки исключений. Обработка исключений и ошибок относится к сквозной функциональности. Она должна реализовываться в отдельных компонентах, которые обеспечивают централизацию этой функциональности и делают ее

доступной во всех слоях приложения. Это также упрощает обслуживание и способствует повторному использованию.

- Протоколирование исключений. Очень важно протоколировать ошибки на границах системы, чтобы служба поддержки могла выявлять и диагностировать их. Это важно для компонентов представления, но может создавать большие сложности для кода, выполняющегося на клиентских компьютерах. Будьте осторожны и тщательно выбирайте методы протоколирования информации личного порядка (Personally Identifiable Information, PII) или конфиденциальных данных и обратите особое внимание на размер и размещение журнала.
- Вывод на экран понятных пользователю сообщений. При использовании этой стратегии в случае возникновения ошибки на экран выводится понятное пользователю сообщение с указанием причины ошибки и описанием, как ее можно исправить. Например, ошибки проверки данных должны отображаться так, чтобы было понятно, какие данные являются ошибочными и почему. В этом сообщении также указывается, как пользователь может исправить или ввести действительные данные.
- Разрешение повторной попытки. При использовании этой стратегии на экран выводится понятное пользователю сообщение, объясняющее причину ошибки и предлагающее пользователю повторить операцию. Такая стратегия полезна, если ошибки формируются из-за возникновения временных исключительных ситуаций, таких как недоступность ресурса или истечение времени ожидания сети.
- Вывод на экран универсальных сообщений. Если в приложении возникает непредвиденная ошибка, данные ошибки необходимо запротоколировать, но для пользователя вывести только универсальное сообщение. Предоставьте пользователю уникальный код ошибки, который может быть представлен группе технической поддержке. Эта стратегия полезна при возникновении непредвиденных исключений. Как правило, в случае возникновения непредвиденного исключения рекомендуется закрыть приложение, чтобы предотвратить повреждение данных или риски безопасности.

Шаг 7 - Определение стратегии валидации

Эффективная стратегия валидации пользовательского ввода поможет фильтровать нежелательные или злонамеренные данные и будет способствовать повышению защищенность приложения.

Как правило, валидация ввода осуществляется слоем представления, тогда как проверка на соответствие бизнес-правилам проводится компонентами бизнес-слоя.

При проектировании стратегии проверки, прежде всего, необходимо определить все вводимые данные, подлежащие проверке. Например, ввод от Веб-клиента в поля формы, параметры (такие как данные операций GET и POST и строки запросов), скрытые поля и состояние представления (view state)

подлежат проверке. В общем, проверяться должны все данные, поступающие из источников, не имеющих доверия.

Для приложений, имеющих компоненты и на стороне клиента, и на стороне сервера, таких как приложения RIA или насыщенные клиентские приложения, вызывающие сервисы на сервере приложений, кроме всех проверок на клиенте, должна проводиться дополнительная проверка на сервере. Но для повышения удобства использования и по соображениям производительности некоторые из проверок можно продублировать на клиенте. Проверка на клиенте позволяет обеспечивать пользователям быструю обратную связь в случаях ввода ими некорректных данных. Это может сохранить время и полосу пропускания, но не забывайте, что злоумышленники могут обойти любую реализованную на клиенте проверку.

Определившись с данными, подлежащими проверке, выберите методики проверки для них. Самыми распространенными методиками проверки являются:

- Прием заведомо допустимого (Список разрешенного ввода или позитивная проверка). Принимаются только данные, удовлетворяющие заданным критериям, все остальные данные отклоняются.
- Отклонение заведомо недопустимого (Список запрещенного ввода или негативная проверка). Принимаются данные, не содержащие известный набор символов или значений.
- Очистка. Известные плохие символы или значения устраняются или преобразовываются с целью сделать ввод безопасным.

Рекомендуется принимать заведомо допустимые значения (Список разрешенного ввода), а не пытаться выявить все возможные недействительные или злонамеренные значения, которые должны быть отклонены. Если невозможно полностью определить список известных допустимых значений, можно дополнить проверку частичным списком известных недопустимых значений и/или проводить очистку в качестве второй линии защиты.

Разные технологии представления используют разные подходы к проверке и информированию пользователя о проблемах. В WPF, к примеру, используются конвертеры и объекты правил проверки, часто подключаемые с помощью XAML, тогда как Windows Forms обеспечивает события проверки и привязки.

2. 7 Проектирование компонентов бизнес - слоя[6]

Проектирование компонентов бизнес-слоя является важной задачей: их неудачный дизайн, скорее всего, впоследствии приведет к созданию кода, который будет сложно обслуживать или расширять.

При проектировании и реализации приложений используется несколько типов компонентов бизнес-слоя. К этим компонентам относятся компоненты бизнес-логики, бизнес-сущности, компоненты бизнес-процесса или рабочего процесса и служебные или вспомогательные компоненты.

[6] Практическое занятие № 3

Шаг 1 - Выбор компонентов бизнес-слоя, которые будут использоваться в приложении

Для реализации бизнес логики в бизнес-слое может понадобиться создать или использовать разные типы компонентов.

Цель этого этапа – выявить эти компоненты и выбрать необходимые для приложения.

Следующие рекомендации помогут принять решение о том, какие компоненты использовать:

- Используйте компоненты бизнес-логики для инкапсуляции бизнес-логики и состояния приложения.

Бизнес-логика – это логика приложения, занимающаяся вопросами реализации бизнес-правил и поведения приложения и обеспечением общей согласованности процессов, таких как валидация данных. Компоненты бизнес-логики должны быть легко тестируемыми и не зависеть от слоев представления и доступа к данным приложения.

- Используйте бизнес-сущности как часть подхода моделирования предметной области для инкапсуляции бизнес-логики и состояния в компоненты, представляющие реальные бизнес-сущности предметной области, такие как продукты и заказы, с которыми должно работать создаваемое приложение.
- Используйте компоненты рабочего процесса, если приложение должно поддерживать многошаговые процессы, выполняемые в определенном порядке; использует бизнес-правила, требующие взаимодействия между многими компонентами бизнес-логики; или вы желаете изменить поведение приложения, обновляя рабочий процесс по мере доработки приложения или изменения требования.

Также рассмотрите возможность использования компонентов рабочего процесса, если приложение должно реализовывать динамическое поведение на основании бизнес-правил. В этом случае сохраняйте правила в обработчике правил. Для реализации компонентов рабочего процесса применяйте Windows Workflow Foundation. В качестве альтернативного варианта можно рассмотреть серверную среду интеграции, такую как BizTalk Server, если приложение должно обрабатывать многошаговый процесс, зависящий от внешних ресурсов, или включает процесс, который должен выполняться как длительная транзакция.

Шаг 2 - Принятие ключевых решений по компонентам бизнес- слоя

То, какие компоненты бизнес-слоя будут использоваться для обработки запросов, определяет общий дизайн и тип создаваемого приложения. Например, компоненты бизнес-слоя Веб-приложения обычно работают с основанными на сообщениях запросами, тогда как приложение Windows Forms обычно взаимодействует с компонентами бизнес-слоя напрямую с помощью основанных на событиях запросов. Кроме того, существуют и другие факторы, которые необходимо учесть при работе с разными типами приложений. Некоторые из этих факторов являются общими для различных типов, тогда как некоторые характерны лишь для конкретного типа приложений.

Рассмотрим ключевые решения, которые должны быть приняты при проектировании компонентов бизнес-слоя:

– Размещение.

Компоненты бизнес-слоя будут размещаться на клиенте, на сервере приложений или и там, и там? Размещайте часть или все компоненты бизнес-слоя на клиенте, если создаете изолированный насыщенный клиент или насыщенное Интернет-приложение (RIA), если желаете улучшить производительность, либо если используете при проектировании бизнес-сущностей модель предметной области. Размещайте часть или все компоненты бизнес-слоя на сервере приложений, если общая бизнес-логика должна поддерживать множество типов клиентов, если компоненты бизнес-слоя требуют доступа к ресурсам, недоступным с клиента, или по соображениям безопасности.

– Связывание.

Как компоненты представления будут взаимодействовать с компонентами бизнес-слоя? Должно ли использоваться тесное связывание, при котором компоненты представления напрямую взаимодействуют с компонентами бизнес-слоя, или слабое связывание, когда применяется абстракция, скрывающая детали компонентов бизнес-слоя? Для простоты в насыщенном клиентском приложении или RIA, в котором оба набора компонентов располагаются на клиенте, можно использовать тесное связывание между компонентами представления и бизнес-слоя, но слабое связывание между этими компонентами обеспечит лучшую тестируемость и гибкость. Если компоненты бизнес-слоя насыщенного клиентского приложения или RIA располагаются на сервере приложений или Веб-сервере, спроектируйте интерфейс сервиса, чтобы обеспечить предельно слабое связывание.

– Взаимодействие.

Если компоненты бизнес-слоя и слоя представления размещаются на одном уровне, используйте компонентные взаимодействия через события и методы, что обеспечивает максимальную производительность. Однако реализуйте интерфейс сервиса и используйте взаимодействия посредством обмена сообщениями между слоем представления и компонентами бизнес-слоя, если компоненты бизнес-слоя и Веб-сервер располагаются на разных уровнях; если разрабатывается Веб-приложение со слабым связыванием между слоем представления и бизнес-слоем; или при наличии насыщенного клиента или приложения RIA. Если насыщенное клиентское приложение или RIA подключаются к серверу приложений или Веб-серверу лишь время от времени, необходимо внимательно подойти к вопросу проектирования интерфейса сервиса, который должен обеспечивать повторную синхронизацию клиента при подключении.

При реализации взаимодействия посредством сообщений продумайте, как будут обрабатываться повторяющиеся запросы и обеспечиваться гарантированная доставка сообщений.

Идемпотентность (способность игнорировать дублирующиеся запросы) важна для сервисных приложений; основанных на сообщениях приложений,

которые используют такую систему обмена сообщениями, как Microsoft Message Queuing; или для Веб-приложений, в которых долго выполняющиеся процессы могут привести к попыткам пользователя выполнить одно действие несколько раз.

Гарантированная доставка необходима для основанных на сообщениях приложений, которые используют такую систему обмена сообщениями, как Microsoft Message Queuing; сервисов, применяющих маршрутизаторы сообщений между клиентом и сервисом; или сервисов, поддерживающих операции типа «отправил и забыл», при которых клиент отправляет сообщение, не ожидая ответа на него. Также не забывайте, что кэшированные сообщения, сохраняемые в ожидании обработки, могут устаревать.

Шаг 3 - Выбор соответствующей поддержки транзакций

Компоненты бизнес-слоя отвечают за координирование и управление всеми транзакциями, которые могут потребоваться в бизнес-слое. Но, прежде всего, необходимо убедиться в необходимости поддержки транзакций.

Транзакции гарантируют, что наборы действий над одним или более диспетчерами ресурсов, такими как базы данных или очереди сообщений, выполняются единым блоком независимо от других транзакций. Если хотя бы одно из действий набора дает сбой, для всех остальных действий должен быть выполнен откат, чтобы обеспечить согласованность состояния системы. Например, имеется операция, которая обновляет три разные таблицы, использующие множество компонентов бизнес-логики. Если два из этих обновлений выполняются успешно, но одно дает сбой, источник данных оказывается в несогласованном состоянии, т.е. содержит некорректные данные, от которых могут зависеть другие операции. Доступны следующие варианты реализации транзакций:

- System.Transactions использует компоненты бизнес-логики для запуска и управления транзакциями. Было введено в версии 2.0 .NET Framework вместе с легковесным диспетчером транзакций (Lightweight Transaction Manager, LTM), используется с диспетчерами недолгосрочных ресурсов или одним диспетчером долгосрочных ресурсов. Этот подход требует явного использования объекта типа TransactionScope (Область действия транзакции) и может расширять область действия транзакции и выполнять делегирование к распределенному координатору транзакций (Distributed Transaction Coordinator, DTC) в случае, если в транзакции участвует несколько диспетчеров долгосрочных ресурсов. Используйте System.Transactions при реализации поддержки транзакций в создаваемом приложений, если имеются транзакции, охватывающие несколько диспетчеров недолгосрочных ресурсов.
- Транзакции WCF были представлены в версии 3.0 .NET Framework и базируются на функциональности System.Transactions. Они обеспечивают декларативный подход к управлению транзакциями, который реализуется посредством ряда атрибутов и свойств, таких как TransactionScopeRequired (Необходима область действия транзакции),

TransactionAutoComplete (Автоматическое завершение транзакции) и TransactionFlow (Поток транзакции). Используйте транзакции WCF, если требуется поддерживать транзакции при взаимодействии с сервисами WCF. Однако рассмотрите возможность применения декларативного описания транзакций, вместо того чтобы использовать код для управления транзакциями.

- Транзакции ADO.NET появились в версии 1.0 .NET Framework, требуют применения компонентов бизнес-логики для запуска и управления транзакциями. Они используют явную модель программирования, когда разработчики должны реализовывать управление нераспределенными транзакциями в коде. Используйте транзакции ADO.NET при расширении приложения, которое уже использует транзакции ADO.NET; или если используете для доступа к базе данных ADO.NET-поставщиков, и транзакции осуществляются только с одним ресурсом. ADO.NET 2.0 и последующие версии дополнительно поддерживают распределенные транзакции, использующие возможности System.Transactions, описанные выше.
- Транзакции базы данных используются для реализации функциональности управления транзакциями, которая может быть включена в хранимые процедуры, что также может упростить дизайн бизнес-процесса. Если транзакции запускаются компонентами бизнес-логики, транзакция базы данных будет входить в состав транзакции, созданной бизнес-компонентом. Используйте транзакции базы данных при разработке хранимых процедур, инкапсулирующих все изменения, которые должны выполняться транзакцией; или при наличии множества приложений, использующих одинаковые хранимые процедуры, когда требования к транзакциям могут быть инкапсулированы в эти хранимые процедуры.

На забывайте, что при использовании распределенных транзакций может увеличиваться связанность между подсистемами системы. Транзакции, включающие удаленные системы, скорее всего, повлекут снижение производительности из-за увеличения сетевого трафика. Транзакции - ресурсоемкий процесс, поэтому они должны выполняться быстро, в противном случае чрезмерно долгая блокировка ресурсов может привести к истечению времени ожидания или взаимным блокировкам.

Участвовать в транзакциях могут только сервисы с высоким уровнем доверия, потому что участие в транзакции позволяет внешним сервисам блокировать внутренние ресурсы. При использовании сервисов для выполнения бизнес-процессов создавайте атомарные транзакции только в крайних случаях, когда этого никак нельзя избежать.

Шаг 4 - Выработка стратегии обработки бизнес-правил

Обработка бизнес-правил может быть одним из наиболее сложных аспектов проектирования приложения. Общей рекомендацией является реализация бизнес-правил в рамках бизнес-слоя. Однако где именно в

бизнес-слое это должно происходить? Это может быть бизнес-логика или компоненты рабочего процесса, обработчик бизнес-правил или дизайн модели предметной области, инкапсулирующий правила в модели.

Рассмотрим возможные варианты обработки бизнес-правил:

- Компоненты бизнес-логики могут использоваться для обработки простых или очень сложных правил в зависимости от шаблона проектирования, применяемого для их реализации. Используйте компоненты бизнес-логики для задач или операций с документами в Веб-приложениях или сервисах, если не реализуете модель предметной области при проектировании бизнес-сущностей, или используете бизнес-правила из внешнего источника.
- Компоненты рабочего процесса используются, если необходимо отделить бизнес- правила от бизнес-сущностей, или если применяемые бизнес-сущности не поддерживают инкапсуляцию бизнес-правил, или если взаимодействие множества бизнес-сущностей управляется инкапсулированной бизнес-логикой.
- Обработчики бизнес-правил обеспечивают возможность задавать и изменять правила без участия разработчиков, но при этом усложняют и добавляют издержки в приложения, поэтому должны использоваться только в случае необходимости. Иначе говоря, применяйте обработчик правил при наличии правил, которые должны будут корректироваться на основании разных факторов, связанных с приложением. Используйте обработчик бизнес-правил при наличии часто меняющихся бизнес-правил, т.е. таких, которые будут меняться регулярно; для поддержки возможности настройки и гибкости; или если хотите предоставить бизнес-пользователям возможность управлять и обновлять правила. Убедитесь, что пользователям предоставляются только те правила, которые подлежат изменениям, и что изменять правила, критические с точки зрения корректности поведения бизнес-логики, могут только авторизованные пользователи.
- Проектирование модели предметной области может использоваться для инкапсуляции бизнес-правил в бизнес-сущности. Но модель предметной области может быть сложно правильно реализовать, кроме того, она имеет тенденцию сосредотачиваться на одном конкретном срезе или контексте. Инкапсулируйте правила в модель предметной области, если имеете насыщенное клиентское приложение или RIA, в котором части бизнес-логики развернуты на клиенте, и сущности модели предметной области инициализируются и сохраняются в памяти; или если имеете модель предметной области, которая может сохраняться в состоянии сеанса, ассоциированном с Веб-приложениями или приложениями сервисов. При размещении частей модели предметной области на клиенте необходимо продублировать ее на сервере, чтобы применить правила и поведение и обеспечить безопасность и возможность обслуживания.

Шаг 5 - Выбор шаблонов, соответствующих требованиям

Поведенческие шаблоны создаются на базе наблюдений за поведением системы в действии и выявлении повторяющихся процессов.

Для компонентов бизнес-слоя обычно используются поведенческие шаблоны проектирования, т.е. шаблоны, основной задачей которых является реализация поведения приложения на уровне дизайна. Необходимо хорошо разбираться в разных типах шаблонов и уметь находить в сценарии поведение, которое может быть описано шаблоном.

В таблице 2.5 приведены шаблоны, которые обычно используются с компонентами бизнес-слоя.

Таблица .2.5

Шаблон	**Рекомендации**
Adapter (Адаптер)	Обеспечивает возможность совместной работы классов с несовместимыми интерфейсами, позволяя разработчикам реализовывать наборы полиморфных классов, обеспечивающих альтернативные реализации существующего класса.
Command (Команда)	Рекомендуется для насыщенных клиентских приложений с меню, панелями инструментов и реализациями клавишных комбинаций быстрого вызова, которые используются для выполнения одних и тех же команд из разных компонентов. Также может использоваться для реализации команд с шаблоном Supervising Presenter.
Chain of Responsibility (Цепочка обязанностей)	Объединяет обработчики запросов так, что каждый обработчик проверяет запрос и либо обрабатывает его, либо передает следующему обработчику. Альтернативой этому шаблону являются выражения «if, then, else» с возможностью обработки сложных бизнес-правил.
Decorator (Декоратор)	Расширяет поведение объекта во время выполнения, добавляя или изменяя операции, которые будут осуществляться при обработке запроса. Требует общего интерфейса, реализовываемого классами декоратора, которые могут объединяться для обработки сложных бизнес-правил.
Dependency Injection (Внедрение зависимостей)	Создает и заполняет члены (поля и свойства) объектов, используя отдельный класс, который обычно создает эти зависимости во время выполнения на основании конфигурационных файлов. Конфигурационные файлы описывают контейнеры, определяющие сопоставление или регистрации типов объектов. Сопоставление и регистрация объектов может также выполняться в коде приложения. Обеспечивает гибкий подход к изменению поведения и реализации сложных бизнес-правил.
Fagade (Фасад)	Обеспечивает слабо детализированные операции,

Шаблон	Рекомендации
	унифицирующие результаты, поступающие от множества компонентов бизнес-логики. Обычно реализуется как удаленный фасад для интерфейсов на основе сообщений бизнес-слоя и используется для обеспечения слабого связывания между слоем представления и бизнес-слоем.
Factory (Фабрика)	Создает экземпляры объектов без указания конкретного типа. Требует наличия объектов, которые реализуют общий интерфейс или расширяют общий базовый класс.
Transaction Script (Сценарий транзакции)	Рекомендуется для базовых CRUD-операций с минимальным набором бизнес- правил. Компоненты сценария транзакции также инициируют транзакции. Это означает, что все операции, осуществляемые компонентом, должны представлять неделимую единицу работы. При использовании этого шаблона компоненты бизнес-логики взаимодействуют с другими компонентами бизнес- слоя и компонентами данных для завершения операции.

Основная задача при выборе шаблона - убедиться, что он соответствует сценарию и не усложняет приложение больше, чем требуется.

2.8 Проектирование компонентов данных[7]

Компонентами слоя доступа к данным являются компоненты, обеспечивающие функциональность доступа к данным, размещаемым в системе, и компоненты агентов сервисов, обеспечивающие функциональность доступа к данным, которые предоставляются другими серверными системами через Веб-сервисы. Кроме того, слой доступа к данным также может включать компоненты, обеспечивающие вспомогательные функции и утилиты.

Шаг 1 - Выбор технологии доступа к данным

При выборе технологии доступа к данным необходимо учесть тип данных, с которыми предполагается работать, и то, как эти данные будут обрабатываться в приложении. Для каждого конкретного сценария есть наиболее подходящие технологии. Чтобы правильно выбрать технологию доступа к данным, соответствующую сценариям создаваемого приложения, руководствуйтесь следующими рекомендациями:

- ADO.NET Entity Framework.

Используйте ADO.NET Entity Framework (EF), если хотите создать модель данных и соотнести ее с реляционной базой данных; соотнести один класс с множеством таблиц, используя наследование; или выполнять запросы к реляционным хранилищам, не входящим в семейство продуктов Microsoft SQL Server. EF подойдет, если имеется объектная модель, которую необходимо соотнести с реляционной моделью, используя гибкую схему, и необходима

[7] Практическое занятие № 4

гибкость, обеспечивающая возможность отделения схемы сопоставления от объектной модели. При использовании EF также рассмотрите возможность применения LINQ to Entities. Используйте LINQ to Entities, если необходимо выполнять запросы через строго типизированные сущности или запрашивать реляционные данные, используя синтаксис LINQ.

– ADO.NET Data Services Framework.

ADO.NET Data Services построена на базе EF и позволяет предоставлять части модели сущностей (Entity Model) через REST- интерфейс. Используйте ADO.NET Data Services Framework, если разрабатываете RIA или n-уровневое насыщенное клиентское приложение и хотите выполнять доступ к данным через ресурсно-ориентированный интерфейс сервиса.

– ADO.NET Core.

Используйте ADO.NET Core, если для обеспечения полного управления доступом к данным в приложении необходим низкоуровневый API; если хотите использовать уже сделанные инвестиции в ADO.NET-решения; если используете традиционную логику доступа к данным. ADO.NET Core подойдет, если нет необходимости в дополнительной функциональности, предлагаемой другими технологиями доступа к данным, или если разрабатываемое приложение должно поддерживать сценарии доступа к данным без постоянного подключения.

– ADO.NET Sync Services.

Используйте ADO.NET Sync Services при проектировании приложения, которое должно поддерживать сценарии без постоянного подключения или требует синхронизации баз данных.

– LINQ to XML.

Используйте LINQ to XML, если приложение работает с XML-данными, запросы к которым необходимо выполнять, применяя синтаксис LINQ.

Шаг 2 - Принятие решения о методе извлечения и хранения бизнес-объектов источника данных

Определившись с требованиями источника данных, необходимо выбрать стратегию заполнения бизнес-объектов или бизнес-сущностей данными из хранилища данных и сохранения данных бизнес-объектов или бизнес-сущностей в хранилище данных.

Обычно интерфейсы объектно-ориентированной модели данных и реляционного хранилища данных не согласованы, что порой усложняет передачу данных между ними. Существует ряд подходов к решению этой проблемы, которые отличаются между собой используемыми типами данных, структурой, технологиями обеспечения транзакций и способами обработки данных.

Самые распространенные подходы используют инструменты и инфраструктуры объектно-реляционного сопоставления (Object/Relational Mapping, O/RM). Используемый в приложении тип сущности является основным фактором при принятии решения о способе сопоставления сущностей со структурами источника данных.

Следующие рекомендации помогут выбрать технику извлечения и сохранения бизнес-объектов в хранилище данных:

- Используйте инфраструктуру O/RM, обеспечивающую преобразования между сущностями предметной области и базы данных. При работе в среде «greenfield», когда вы имеете полный контроль над схемой базы данных, инструмент O/RM может обеспечить формирование схемы для поддержки объектной модели и сопоставления сущностей базы данных и предметной области. При работе в среде «brownfield», где вам приходится использовать предлагаемую схему базы данных, инструмент O/RM поможет сопоставить модель предметной области и реляционную модель.
- В объектно-ориентированном проектировании обычно используется модель предметной области, шаблон, основанный на моделировании сущностей соответственно объектам предметной области.
- Правильно сгруппируйте сущности, чтобы достичь высокого уровня связности. Для этого может понадобиться ввести в модель предметной области дополнительные объекты и сгруппировать взаимосвязанные сущности в сводные корни.
- При работе с Веб-приложениями или сервисами группируйте сущности и обеспечивайте опции для частичной загрузки сущностей предметной области только необходимыми данными. Это сократит использование ресурсов за счет того, что не придется удерживать в памяти инициализированные модели предметной области для каждого пользователя, и позволит приложениям справляться с более высокой нагрузкой пользователей.

Шаг 3 - Выбор способа подключения к источнику данных

Зная, как компоненты доступа к данным сопоставляются с источником данных, можно принять решение о том, как будет выполняться подключение к источнику данных, реализовываться защита пользовательских учетных данных и выполняться транзакции.

Подключения

Подключения к источникам данных - это фундаментальная часть слоя доступа к данным. Слой доступа к данным должен координировать все подключения к источнику данных, используя для этого инфраструктуру доступа к данным. На создание и управление подключениями приходится расходовать ценные ресурсы и в слое доступа к данным, и в самом источнике данных.

Следующие рекомендации помогут выбрать соответствующую технику подключения к источникам данных:

- Открывайте подключения к источнику данных как можно позже и закрывайте их как можно раньше. Это обеспечит блокировку ресурсов лишь на короткие промежутки времени и сделает их более доступными для других процессов. Для малоизменяющихся данных используйте оптимистическую блокировку, это снизит издержки на блокировку строк

базы данных, включая затраты на подключение, которое должно оставаться открытым в течение блокировки.

- Там где возможно, осуществляйте транзакции через одно подключение. Это позволит использовать возможности транзакций ADO.NET без внешних сервисов координации распределенных транзакций.
- Используйте пул подключений и оптимизируйте производительность на основании результатов нагрузочных тестов. Рассмотрите возможность настройки уровней изоляции подключений для запросов к данным. В приложении с высокими требованиями к пропускной способности некоторые операции с данными могут выполняться на более низких уровнях изоляции, чем остальные операции транзакции. Сочетание уровней изоляции может иметь негативное влияние на согласованность данных, поэтому этот вариант необходимо тщательно анализировать для каждого конкретного случая в отдельности.
- По соображениям безопасности избегайте использования системных или пользовательских DSN для хранения данных подключения.
- Предусмотрите логику повторного подключения для случаев разрыва соединения с источником данных или его закрытия по истечении времени ожидания.
- По возможности используйте пакетные команды, что позволит сократить количество обращений к серверу базы данных.

Другой важный аспект, который необходимо учесть - требования к безопасности в связи с доступом к источнику данных. Иначе говоря, необходимо продумать, как источник данных будет аутентифицировать компоненты доступа к данным, и каковы будут требования к авторизации. Следующие рекомендации обеспечат проектирование безопасного подхода для подключения к источникам данных:

- Предпочтительнее использовать аутентификацию Windows, а не аутентификацию SQL Server. При работе с Microsoft SQL Server используйте аутентификацию Windows с доверенной подсистемой.
- При использовании аутентификации SQL применяйте учетные записи с надежными паролями; с помощью ролей базы данных ограничьте права доступа каждой учетной записи в рамках SQL Server; добавьте ACL во все файлы, используемые для хранения строк подключения; и шифруйте строки подключения в конфигурационных файлах.
- Используйте учетные записи с наименьшими правами доступа к базе данных и требуйте от вызывающей стороны предоставлять слою данных идентификационные данные для целей аудита.
- Не храните пароли для проверки пользователей в базе данных, ни в виде открытого текста, ни в зашифрованном виде. Храните хеши паролей с шумом (случайные разряды, используемые как один из параметров в функции хеширования).
- При использовании SQL-выражений для доступа к источнику данных четко обозначьте границы доверия и применяйте параметризованные

запросы, а не конкатенацию строк. Это обеспечит защиту от атак через внедрение SQL-кода.

- Защитите конфиденциальные данные, передаваемые по сети к и от SQL Server. Не забывайте, что аутентификация Windows обеспечивает защиту учетных данных, но не данных приложения. Для защиты данных в канале передачи используйте протоколы IPSec или SSL.

Пул подключений

Пул подключений обеспечивает возможность повторного использования приложением подключения из пула или создания нового подключения и добавления его в пул в случае недоступности подходящего подключения.

Когда приложение закрывает подключение, оно возвращается в пул, и базовое подключение остается открытым. Это означает, что ADO.NET не надо создавать новое подключение к источнику данных, и каждый раз открывать его заново. Хотя использование пула открытых подключений потребляет ресурсы, это обеспечивает сокращение задержек при доступе к данным и повышает эффективность выполнения приложения в случае доступности подходящих соединений из пула. Рассмотрим другие вопросы использования пула подключений:

- Чтобы максимально повысить эффективность пула подключений, используйте модель безопасности доверенная подсистема и по возможности избегайте олицетворения. Использование минимального числа учетных записей повышает вероятность повторного использования подключения из пула и сокращает шансы переполнения пула подключений. Если каждый вызов использует разные учетные данные, ADO.NET приходится каждый раз создавать новое подключение.
- Подключения, которые остаются открытыми в течение длительного периода времени, могут удерживать ресурсы на сервере. Обычная причина этого - раннее открытие подключений и позднее их закрытие (например, когда подключение не закрывается явно и не удаляется до тех пор, пока не выходит за рамки области действия).
- Подключения могут оставаться открытыми в течение длительного времени при использовании объектов DataReader, которые являются действительными только пока подключение открыто.

Транзакции и параллелизм

При наличии в приложении ответственных операций используйте транзакции для их выполнения.

Транзакции позволяют выполнять связанные действия с базой данных как единую операцию и гарантировать тем самым целостность базы данных. Транзакция считается завершенной, если все входящие в нее действия выполнены, после этого внесенные в ходе этой транзакции изменения базы данных становятся постоянными. Транзакции поддерживают отмену (откат) действий в случае возникновения ошибки, что помогает сохранить целостность данных в базе данных. Следующие рекомендации помогут при проектировании транзакций:

- При проектировании доступа к одному источнику данных по возможности используйте транзакции на базе подключения. При использовании создаваемых вручную или явных транзакций реализуйте транзакцию в хранимой процедуре. Если не можете использовать транзакции, реализуйте компенсационные методы для возвращения хранилища данных в предыдущее состояние.
- При использовании длительных атомарных транзакций избегайте слишком долгого удержания блокировок. В подобных сценариях лучше использовать компенсационные блокировки. Если для завершения транзакции требуется длительное время, используйте асинхронные транзакции, осуществляющие обратный вызов клиента по завершении. Также для параллельно выполняющихся приложений, осуществляющих большое число транзакций, используйте технологию MARS (множество активных результирующих множеств), это позволит избежать потенциальных взаимоблокировок.
- Если вероятность возникновения конфликта данных из-за их одновременного изменения несколькими пользователями низка (например, когда пользователи, преимущественно, добавляют данные или редактируют разные строки), используйте оптимистическую блокировку, при которой действительным считается последнее обновление. Если вероятность возникновения конфликта данных из-за их одновременного изменения несколькими пользователями высока (например, когда пользователи, преимущественно, редактируют одни и те же строки), используйте пессимистическую блокировку, при которой обновление может применяться только к последней версии данных. Также учтите вопросы параллельной обработки при доступе к статическим данным приложения или при использовании потоков для осуществления асинхронных операций. Статические данные по природе своей не являются потокобезопасными, т.е. изменения, вносимые в такие данные в одном потоке, будут оказывать влияние на другие потоки, использующие эти же данные.
- Транзакции должны быть максимально короткими, это обеспечит самые короткие блокировки и улучшит условия параллельной работы. Однако не следует забывать, что короткие и простые транзакции могут привести к созданию слишком детализированного интерфейса, которому для завершения одной операции понадобится делать множество вызовов.
- Используйте соответствующий уровень изоляции. Необходимо найти баланс между непротиворечивостью данных и конкуренцией за ресурсы. Более высокий уровень изоляции обеспечит более высокую непротиворечивость данных за счет общего снижения возможностей для параллельной обработки. Более низкий уровень изоляции, снижая конкуренцию за ресурсы, улучшает производительность ценой потери непротиворечивости данных.

Существует три общих типа поддержки транзакций:

– Классы пространства имен System.Transactions обеспечивают поддержку явных и неявных транзакций как часть .NET Framework.

Используйте System.Transactions при разработке нового приложения, требующего поддержку транзакций, или при наличии транзакций, охватывающих несколько диспетчеров недолгосрочных ресурсов. Для реализации большинства транзакций рекомендуется использовать явную модель, которую обеспечивает объект TransactionScope пространства имен System.Transactions. Хотя неявные транзакции не настолько быстрые, как созданные вручную, или явные, но их проще создавать, и они обеспечивают решения промежуточного уровня, гибкие и более простые в обслуживании. Если не желаете использовать неявную модель для транзакций, можно реализовать создание транзакций вручную, используя класс Transaction пространства имен System.Transactions.

– Транзакции ADO.NET, использующие единственное подключение к базе данных.

Это наиболее эффективный подход для управляемых клиентом транзакций с одним хранилищем данных. Выбирайте транзакции ADO.NET, если расширяете приложение, уже использующее транзакции ADO.NET; если используете поставщиков ADO.NET для доступа к базе данных и транзакции выполняются только к одной базе данных; или если развертываете приложение в среде, не поддерживающей версию 2.0 .NET Framework. Команды ADO.NET обеспечивают начало, фиксацию и откат операций, осуществляемых в рамках транзакции.

– Транзакции T-SQL (базы данных), управляемые командами, выполняемыми в базе данных.

Это наиболее эффективный способ реализации управляемых сервером транзакций с одним хранилищем данных, при котором база данных контролирует все аспекты транзакции. Используйте транзакции базы данных при разработке хранимых процедур, инкапсулирующих все изменения, которые должны быть выполнены транзакцией, или при наличии множества приложений, использующих одни и те же хранимые процедуры, когда требования транзакции могут быть инкапсулированы в хранимые процедуры.

Шаг 4 - Выработка стратегий обработки ошибок источника данных

На данном этапе должна быть выработана общая стратегия обработки ошибок источников данных.

Все исключения, связанные с источниками данных, должны перехватываться слоем доступа к данным.

Исключения, касающиеся самих данных, а также ошибки доступа к источнику данных и истечения времени ожидания, должны обрабатываться в этом слое и передаваться в другие слои, только если эти сбои оказывают влияние на время отклика или функциональность приложения.

Исключения

Стратегия централизованного управления исключениями обеспечит единообразие при обработке исключений. Обработка исключений относится к

сквозной функциональности, поэтому эту логику рекомендуется реализовывать в отдельных компонентах, которые могут использоваться совместно слоями и уровнями приложения.

Особое внимание необходимо уделить исключениям, распространяющимся через границы доверия и на другие слои или уровни, и необрабатываемым исключениям, чтобы они не приводили к нарушению надежности приложения или раскрытию конфиденциальных данных приложения. Следующий подход поможет при проектировании стратегии обработки исключений:

- Определите, какие исключения должны перехватываться и обрабатываться слоем доступа к данным. Проверки на наличие взаимоблокировок, проблем с подключениями и нежестких блокировок обычно могут проводиться в рамках слоя данных.
- Рассмотрите возможность реализации повторных попыток для операций, в которых могут возникать ошибки, связанные с источником данных или с истечением времени ожидания, но только в случаях, когда это безопасно.
- Разрабатывайте соответствующую стратегию распространения исключений. Например, обеспечьте возможность передачи исключений в граничные слои, где они могут быть запротоколированы и преобразованы соответствующим образом для передачи в следующий слой.
- Выработайте соответствующую стратегию протоколирования и уведомления о критических ошибках и исключениях, обеспечивая сокрытие конфиденциальных данных.
- Используйте существующие инструменты, такие как Enterprise Library от группы patterns & practices, для реализации единообразной стратегии обработки и управления исключениями.

Логика повтора попыток

Предусмотрите логику повтора попыток для обработки ошибок, возникающих при переходе на другой ресурс в случае сбоя сервера или базы данных. Логика повтора должна перехватывать все ошибки, возникающие при подключении к базе данных или выполнении команд (запросов или транзакций). Причин формирования ошибки может быть множество. При возникновении ошибки компонент данных должен восстановить подключение, закрыв существующие подключения и создав новое, и затем повторить команды, давшие сбой, если это необходимо. Повторные попытки должны выполняться лишь определенное количество раз, после чего, в случае их неудачи, попытки выполнить команды прекращаются, и возвращается исключение. Все запросы и любые последующие повторные попытки должны выполняться асинхронно, это обеспечит невозможность создания ситуации, когда приложение не отвечает.

Истечение времени ожидания

Очень важно правильно выбрать время ожидания для подключения и команды. Задание времени ожидания для подключения или команды,

превышающего время ожидания клиента (например, в случае со временем ожидания запроса Веб-приложения, браузера или Вебсервера), может привести к тому, что время ожидания запроса клиента истечет до того, как подключение к базе данных будет открыто. Задание недостаточного времени ожидания приведет к тому, что обработчик ошибок начнет выполнять логику повтора попыток. Если время ожидания истекает во время выполнения транзакции, в случае использования пула подключений ресурсы базы данных могут остаться заблокированными после закрытия подключения. В таких случаях, чтобы закрытое подключение не возвращалось в пул, оно должно удаляться. Это обеспечивает откат транзакции и высвобождение ресурсов базы данных.

Шаг 5 - Проектирование объектов агентов сервисов (необязательный)

Агенты сервисов - это объекты, которые управляют семантикой взаимодействия с внешними сервисами, изолируют приложение от специфических особенностей взаимодействия с разными сервисами и обеспечивают дополнительные сервисы, такие как сопоставление формата данных, предоставляемого сервисом, и формата, требуемого приложением. Они также могут реализовывать кэширование и поддержку работы в автономном режиме или неустойчивого подключения. Выполняйте разработку объектов агентов сервисов в следующей последовательности:

1. Используйте соответствующий инструмент для добавления ссылки на сервис. Это обеспечит формирование прокси-классов и классов данных, представляющих контракт данных сервиса.
2. Определите, как сервис будет использоваться в приложении. Для большинства приложений агент сервиса выступает в роли уровня абстракции между бизнес-слоем и удаленным сервисом и может обеспечивать единообразный интерфейс независимо от формата данных. В небольших приложениях слой представления может выполнять доступ к агенту сервиса напрямую.

2.9 Показатели качества

Показатели качества – это общие факторы, оказывающие влияние на поведение во время выполнения, дизайн системы и взаимодействие с пользователем. Они представляют функциональные области, потенциально влияющие на все приложение со всеми его слоями и уровнями.

Некоторые из этих показателей касаются общего дизайна системы, тогда как другие относятся только ко времени выполнения, времени проектирования или связаны только с вопросами взаимодействия с пользователем. Степень, с которой приложение реализует желаемое сочетание показателей качества, таких как удобство и простота использования, производительность, надежность и безопасность, свидетельствует об успешности дизайна и общем качестве программного приложения.

При выполнении любого из требований показателей качества во время проектирования приложений важно учесть, какое влияние это может оказать на другие требования. Необходимо проанализировать соотношение выгод и потерь

для совокупности множества показателей качества. Важность или приоритетность каждого из показателей качества в разных системах разная; например, возможность взаимодействия с другими системами, как правило, не так важна для коробочного приложения индивидуального использования, чем для бизнес-системы (LOB).

Общие показатели качества

В таблице 2.6 показатели качества разбиты на четыре категории, касающиеся качества дизайна, качеств времени выполнения, системы и взаимодействия с пользователем. Используйте данную таблицу, чтобы понять, какое значение имеет каждый из атрибутов с точки зрения дизайна приложения.

Таблица 2.6

Категория	**Показатель качества**	**Описание**
Качества дизайна	Концептуальная целостность	определяет согласованность и связность дизайна в целом. Сюда относится и то, как спроектированы компоненты или модули, и такие факторы как стиль написания кода и именование переменных.
	Удобство и простота обслуживания	способность системы изменяться. Это касается изменения компонентов, сервисов, функций и интерфейсов при добавлении или изменении функциональности, исправлении ошибок и реализации новых бизнес-требований.
	Возможность повторного использования	определяет пригодность компонентов и подсистем к использованию в других приложениях и сценариях. Возможность повторного использования обеспечивает снижение дублирования компонентов и также сокращение времени, затрачиваемого на реализацию.
Качества времени выполнения	Доступность	определяет, какую часть времени система функциональна и работает. Доступность может быть измерена как процентное соотношение времени простоя системы за заданный промежуток времени. На доступность оказывают влияние ошибки системы, проблемы инфраструктуры, злонамеренные атаки и нагрузка системы.

Категория	**Показатель качества**	**Описание**
	Возможность взаимодействия	Способность системы или разных систем успешно работать через взаимодействие и обмен данными с другими внешними системами, созданными и выполняемыми внешними сторонами. Способная к взаимодействию система упрощает обмен и повторное использование данных, как внутри, так и вне ее границ.
	Управляемость	определяет, насколько просто системным администраторам управлять приложением, как правило, посредством достаточного и полезного инструментария, предоставляемого для использования в системах мониторинга, а также для отладки и настройки производительности.
	Производительность	Показатель, характеризующий скорость, с какой система выполняет любое действие в заданный промежуток времени. Производительность измеряется в показателях задержки или пропускной способности. Задержка - это время, необходимое для ответа на любое событие. Пропускная способность - это число событий, имеющих место в заданный промежуток времени.
	Надежность	Способность системы сохранять работоспособность в течение некоторого времени. Надежность определяется как вероятность того, что система сможет выполнять предусмотренные функции в течение заданного промежутка времени.
	Масштабируемость	Способность системы справляться с увеличением нагрузки без влияния на ее производительность или способность легко расширяться.
	Безопасность	Способность системы предотвращать злонамеренные или случайные действия, не предусмотренные при проектировании, или не допускать

Категория	Показатель качества	Описание
		разглашение или утрату данных. Безопасная система должна защищать ресурсы и предотвращать несанкционированные изменения данных.
Качества системы	Обеспеченность технической поддержкой	Способность системы предоставлять сведения, необходимые для выявления и разрешения проблем при некорректной работе.
	Тестируемость	Мера того, насколько просто создать критерий проверки для системы и ее компонентов и выполнить эти тесты, чтобы определить, отвечает ли система данному критерию. Хорошая тестируемость означает большую вероятность того, что сбои в системе могут быть своевременно и эффективно изолированы.
Качества взаимодействия с пользователем	Удобство и простота использования	определяет, насколько приложение соответствует требованиям пользователя и потребителя с точки зрения понятности, простоты локализации и глобализации, удобства доступа для пользователей с физическими недостатками и обеспечения хорошего взаимодействия с пользователем в общем.

2.10 Проектирование общей функциональности приложения

В большинстве проектируемых приложений имеется общая функциональность, которую нельзя отнести к конкретному слою или уровню. Обычно это такие операции как аутентификация, авторизация, кэширование, связь, управление исключениями, протоколирование и инструментирование, а также валидация.

Эти функции называют сквозной функциональностью (crosscutting concerns), потому что они оказывают влияние на все приложение и, по возможности, должны реализовываться централизованно. Например, если код, формирующий записи журнала и выполняющий запись в журналы приложения, разбросан по разным слоям и уровням, в случае изменения требований, связанных с этими вопросами (например, перенесение журнала в другой каталог), придется выискивать и обновлять соответствующий код по всему

приложению. Но если код протоколирования централизован, изменить поведение, можно изменив код лишь в одном месте.

Существует несколько разных подходов к реализации этой функциональности, начиная от общих библиотек, таких как Enterprise Library группы patterns & practices, до методов аспектно-ориентированного программирования (Aspect Oriented Programming, AOP), использующих метаданные для внедрения кода сквозной функциональности непосредственно в откомпилированный вывод или во время выполнения.

Следующие рекомендации помогут понять основные факторы, оказывающие влияние на сквозную функциональность:

- Проанализируйте все функции в каждом слое и найдите те из них, которые могут быть выделены в общие компоненты, возможно, даже компоненты общего назначения, настраиваемые в зависимости от конкретных требований каждого слоя приложения. Скорее всего, эти компоненты можно будет использовать и в других приложениях.
- В зависимости от физического распределения компонентов и слоев приложения, возможно, понадобится установить компоненты сквозной функциональности на нескольких физических уровнях. Но несмотря на это, преимущества от возможности повторного использования и сокращения времени и затрат на разработку сохраняются.
- Используйте шаблон Dependency Injection для внедрения экземпляров компонентов сквозной функциональности в приложение на основании данных конфигурации. Это позволяет без труда изменять используемые в каждой подсистеме компоненты сквозной функциональности без необходимости повторной компиляции и развертывания приложения. Библиотека Unity группы patterns & practices обеспечивает полную поддержку шаблона Dependency Injection. К другим популярным библиотекам Dependency Injection относятся StructureMap, Ninject и Castle Windsor.
- Сократить время разработки позволит использование библиотек компонентов сторонних производителей, предоставляющих легкоконфигурируемые компоненты. Одним из примеров такой библиотеки является Enterprise Library группы patterns & practices, содержащая блоки приложений, которые облегчат реализацию функций кэширования, обработки исключений, аутентификации и авторизации, протоколирования, валидации и шифрования. Она также включает механизмы, реализующие контейнер внедрения политик и зависимостей, которые упрощают реализацию решений для ряда аспектов сквозной функциональности. Другой популярной библиотекой является Castle Project.
- Используйте методики аспектно-ориентированного программирования (AOP), что поможет внедрить сквозную функциональность в приложение без реализации явных вызовов в коде. Библиотека Unity и Enterprise Library Policy Injection Application Block (Блок внедрения

политик библиотеки Enterprise Library) группы patterns & practices поддерживают этот подход. Другими примерами являются библиотеки Castle Windsor и PostSharp .

2.11 Стратегии развертывания приложений[8]

Дизайны архитектуры приложений существуют в виде моделей, документов и сценариев. Однако приложения развертываются в физической среде, ограничения которой могут вносить коррективы в некоторые архитектурные решения. Следовательно, предполагаемые сценарий развертывания и инфраструктура должны рассматриваться как часть процесса проектирования приложения. В данной главе описываются возможные варианты развертывания различных типов приложений, включая распределенное и нераспределенное развертывание, пути масштабирования приложения, а также руководство и подходы к обеспечению производительности, надежности и безопасности. Рассматривая возможные сценарии развертывания приложения как часть процесса его проектирования, вы предотвращаете ситуацию неудачного развертывания приложения или невозможности выполнения тех или иных требований дизайна из-за технических ограничений инфраструктуры.

Выбор стратегии развертывания сопряжен с нахождением компромиссов в дизайне. Они могут быть связаны с ограничениями по использованию протоколов взаимодействия или портов либо особыми топологиями развертывания, не поддерживаемыми целевой инфраструктурой. Выявление ограничений развертывания на ранних этапах проектирования поможет избежать сюрпризов в будущем. Привлекайте к этой работе группы обслуживания сети и инфраструктуры.

При выборе стратегии развертывания:

1. Изучите целевую физическую инфраструктуру развертывания.
2. Исходя из инфраструктуры развертывания, выявите ограничения архитектуры и дизайна.
3. Выявите, какое влияние на безопасность и производительность разрабатываемой системы будет оказывать инфраструктура развертывания.

Распределенное и нераспределенное развертывание

При выработке стратегии развертывания, прежде всего, необходимо определиться, какая модель развертывания будет использоваться: распределенное или нераспределенное развертывание.

Если создается простое приложение для применения во внутренних сетях, подойдет нераспределенное развертывание. Для более сложного приложения, которое должно быть оптимизировано для обеспечения масштабируемости и удобства обслуживания, применяйте распределенное развертывание.

---

[8] Лабораторная работа № 1

Нераспределенное развертывание

При нераспределенном развертывании вся функциональность и слои приложения, кроме функциональности хранения данных, располагаются на одном сервере, как показано на рис. 2.5.

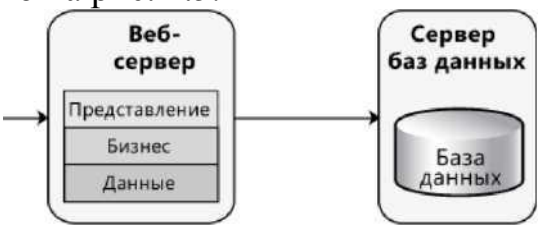

Рис. 2.5 Нераспределенное развертывание

Преимуществом данного подхода является простота и минимальные требования по количеству необходимых физических серверов.

Также обеспечивается наилучшая производительность, поскольку взаимодействие между слоями осуществляется без пересечения физических границ между серверами или кластерами серверов. Но нельзя забывать, что несмотря на сокращение издержек на взаимодействие, использование одного сервера создает другие угрозы производительности. Прежде всего, совместное использование ресурсов всеми слоями приложения. Если один из слоев начинает слишком активно потреблять ресурсы, это негативно сказывается на работе всех остальных слоев. Кроме того, используемые серверы должны быть одинаково конфигурированы и спроектированы с самым строгим соблюдением эксплуатационных требований и должны поддерживать пиковые нагрузки, возникающие при использовании ресурсов системы максимальным числом пользователей. Использование одноуровневой архитектуры снижает общую масштабируемость и удобство обслуживания приложения, поскольку все слои физически располагаются на одном оборудовании.

Распределенное развертывание

При распределенном развертывании все слои приложения располагаются на разных физических уровнях. При многоуровневом развертывании инфраструктура системы организована как набор физических уровней для обеспечения серверных сред, оптимизированных соответственно определенным эксплуатационным требованиям и требованиям по использованию системных ресурсов. Такая модель позволяет распределять слои приложения по разным физическим уровням, как показано в примере на рис. 2.6.

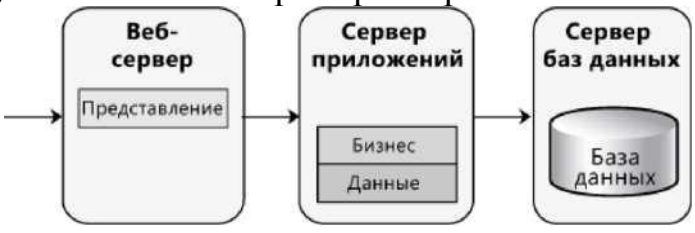

Рис.2.6 Распределенное развертывание

Распределенный подход позволяет конфигурировать серверы приложений для размещения на них различных слоев с целью обеспечения максимального соответствия требованиям каждого слоя. Тем не менее, поскольку основным фактором оптимизации развертывания компонентов является сопоставление

профиля потребления ресурсов компонента с определенным сервером, прямое проецирование слоев в уровни часто не является наилучшей стратегией развертывания.

Многоуровневость обеспечивает возможность использования множества инфраструктур. Каждая инфраструктура оптимизируется под определенный набор эксплуатационных требований и требований по использованию системных ресурсов. После этого компоненты могут быть развернуты на уровне, наиболее точно соответствующем требованиям по ресурсам, что обеспечит наилучшие производительность и поведение. Чем больше уровней, тем больше вариантов развертывания имеется для каждого компонента. Распределенное развертывание обеспечивает более гибкую среду, в которой в случае необходимости намного проще реализовать горизонтальное или вертикальное масштабирование. Однако нельзя забывать, что введение большего количества уровней ведет к повышению сложности, увеличению необходимого объема работ по развертыванию и затрат.

Другое основание для добавления уровней - применение особых политик безопасности. Распределенное развертывание позволяет обеспечить более высокую защиту серверам приложений, например, через введение межсетевого экрана между Веб-сервером и серверами приложений и использование разных вариантов аутентификации и авторизации.

Распределение компонентов по уровням может привести к снижению производительности из-за издержек на удаленные вызовы через физические границы. Однако распределение компонентов может улучшить возможности масштабирования, удобство обслуживания, приводя к снижению затрат в долгосрочной перспективе.

Рекомендации по размещению компонентов при распределенном развертывании

При проектировании распределенного развертывания необходимо, прежде всего, распределить имеющиеся логические слои и компоненты по физическим уровням.

В большинстве случаев слой представления размещается на клиенте или на Веб-сервере; слой сервисов, бизнес-слой и слой доступа к данным - на сервере приложений; и база данных - на собственном сервере. Но такая схема не является обязательной.

При принятии решения о размещении компонентов в распределенной среде руководствуйтесь следующими рекомендациями:

- Распределяйте компоненты только в случае необходимости. Основаниями для реализации распределенного развертывания являются политики безопасности, физические ограничения, совместное использование бизнес-логики и масштабируемость.
- Если бизнес-компоненты используются компонентами представления синхронно, развертывайте их на одном уровне, чтобы обеспечить максимальную производительность и упростить операционное управление.

- Не размещайте компоненты представления и бизнес-компоненты на одном уровне, если аспекты безопасности требуют установления границы доверия между ними. Например, компоненты представления и бизнес-компоненты в насыщенном клиентском приложении можно разделить, разместив компоненты представления на клиенте и бизнес-компоненты на сервере.
- Если аспекты безопасности не требуют установления границы доверия между компонентами агентов сервиса и использующим их кодом, размещайте их на одном уровне.
- По возможности развертывайте вызываемые асинхронно бизнес-компоненты и компоненты рабочего процесса на другом физическом уровне, отдельно от всех остальных слоев приложения.
- Размещайте бизнес-сущности на одном уровне с компонентами, их использующими.

Шаблоны распределенного развертывания

Схемы развертывания, используемые в большинстве решений, могут быть описаны несколькими общими шаблонами. При выборе наилучшего решения развертывания для приложения полезно сначала обозначить общие шаблоны. Полностью разобравшись и поняв разные схемы, можно выбрать наиболее походящую из них, учитывая конкретные сценарии, требования и ограничения безопасности.

Развертывание клиент-сервер

Этот шаблон представляет базовую структуру с двумя основными компонентами: клиент и сервер.

При таком сценарии клиент и сервер обычно размещаются на двух разных уровнях. На рис. 2.7 представлен типовой сценарий для Веб-приложения, в котором клиент взаимодействует с Веб-сервером.

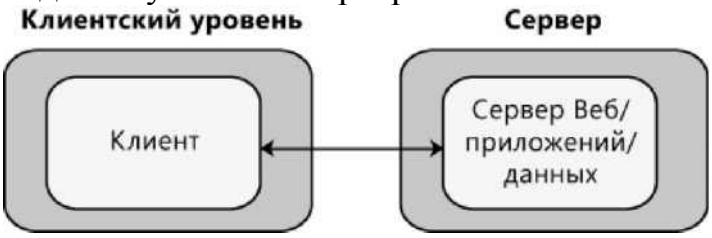

Рис.2.7 Типовой сценарий реализации Веб-приложения

Шаблон клиент/сервер рекомендуется применять при разработке клиента, который будет осуществлять доступ к серверу приложений, или при разработке автономного клиента, который будет взаимодействовать с отдельным сервером базы данных.

n-уровневое развертывание

n-уровневый шаблон представляет общую схему развертывания, при которой компоненты приложения развернуты на одном или более серверах. Чаще всего используются 2-уровневый, 3-уровневый или 4-уровневый шаблон. Обычно все компоненты слоя размещаются на одном уровне, но это не является обязательным правилом. Нет требования по абсолютному совпадению слоев и

уровней, рабочая нагрузка может быть распределена между несколькими серверами в случае необходимости. Например, разные аспекты бизнес-логики могут располагаться на разных уровнях.

2-уровневое развертывание

По сути, физически это та же компоновка, что и шаблон клиента/сервер. Основное отличие в том, как происходит взаимодействие компонентов уровней. В некоторых случаях, как показано на рис. 2.8, весь код приложения размещается на клиенте, и база данных выносится на отдельный сервер. Клиент использует хранимые процедуры или минимальную функциональность доступа к данным сервера базы данных.

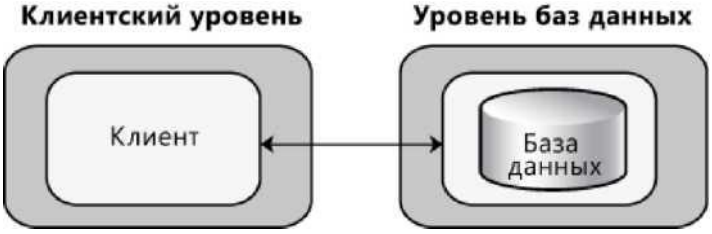

Рис. 2.8 2-уровневое развертывание с размещением всего кода приложения на клиенте

Используйте 2-уровневый шаблон при разработке клиента, который будет осуществлять доступ к серверу приложений, или при разработке автономного клиента, который будет взаимодействовать с отдельным сервером базы данных.

3-уровневое развертывание

В 3-уровневом дизайне клиент взаимодействует с кодом приложения, развернутым на другом сервере, и сервер приложений взаимодействует с базой данных, размещенной на отдельном сервере, как показано на рис. 2.9.

Этот шаблон применяется для большинства Веб-приложений и Веб-сервисов и подходит для многих общих сценариев. Между клиентом и уровнем Веб/приложений, уровнем Веб/приложений и уровнем базы данных могут устанавливаться межсетевые экраны.

Рис 2.9 3-уровневое развертывание с вынесением кода приложения на отдельный уровень

Используйте 3-уровневый шаблон для Интранет-приложения, в котором все серверы размещаются в частной сети, или для Интернет-приложения, для которого требования по обеспечению безопасности не запрещают реализацию бизнес-логики на внешнем Веб-сервере или сервере приложений.

4-уровневое развертывание

В данном сценарии (рис. 2.10) Веб-сервер и сервер приложений физически разделены. Часто это делают из соображений безопасности, когда Веб-сервер

развертывается в пограничной сети и доступен для сервера приложений, размещенного в другой подсети. При таком сценарии между клиентом и Веб-уровнем, а также между Веб-уровнем и уровнем приложений или бизнес-логики могут быть реализованы межсетевые экраны.

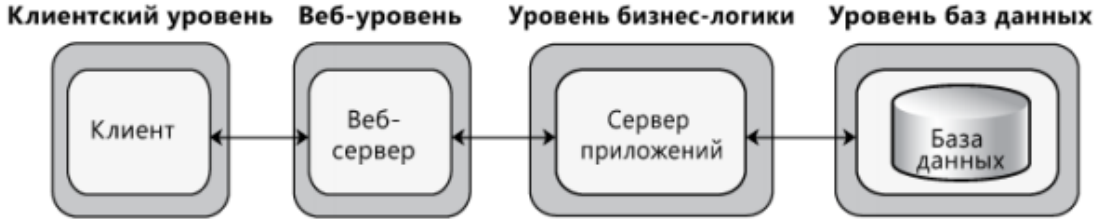

Рис. 2.10 4-уровневое развертывание, при котором код Веб-приложения и бизнес-логика размещаются на разных уровнях

Используйте 4-уровневый шаблон, если по требованиям безопасности бизнес-логика не может размещаться в пограничной сети, или если имеется код приложения, активно использующий ресурсы сервера, и эту функциональность требуется перенести на другой сервер.

Развертывание веб-приложения

Используйте распределенное развертывание для Веб-приложений, если из соображений безопасности бизнес-логика не может быть развернута на внешнем Веб-сервере.

Взаимодействие посредством обмена сообщений и TCP-протокол с бинарным кодированием обеспечит лучшую производительность для бизнес-слоя. Также необходимо продумать балансировку нагрузки для распределения запросов, так чтобы они обрабатывались разными Веб-серверами, это поможет избежать привязки к конкретному серверу при проектировании масштабируемых Веб-приложений. Используйте в Веб-приложении компоненты без сохранения состояния.

Развертывание насыщенного Интернет-приложения

Распределенная архитектура наиболее подходящий сценарий развертывания в реализациях насыщенных Интернет-приложений (RIA), поскольку позволяет переносить логику представления на клиент.

Если бизнес-логика приложения совместно используется другими приложениями, используйте распределенное развертывание. Кроме того, предусмотрите для бизнес-логики интерфейс взаимодействия на основе сообщений.

Развертывание насыщенного клиентского приложения

При n-уровневом развертывании на клиенте могут располагаться логика представления и бизнес-логика или только логика представления. Рис. 2.11 показан вариант, когда на клиенте размещаются логика представления и бизнес-логика.

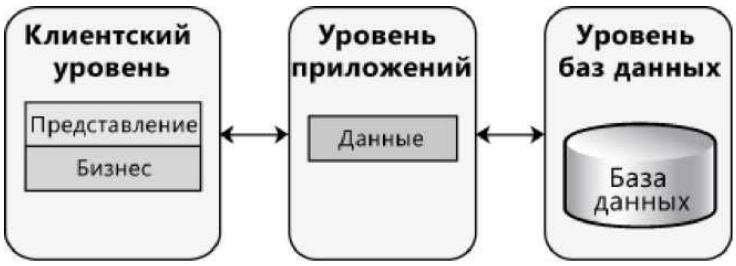

Рис. 2.11 Насыщенный клиент, бизнес-слой которого размещается на клиентском уровне

На рис. 2.12 проиллюстрирован вариант размещения бизнес-логики и логики доступа к данным на сервере приложений.

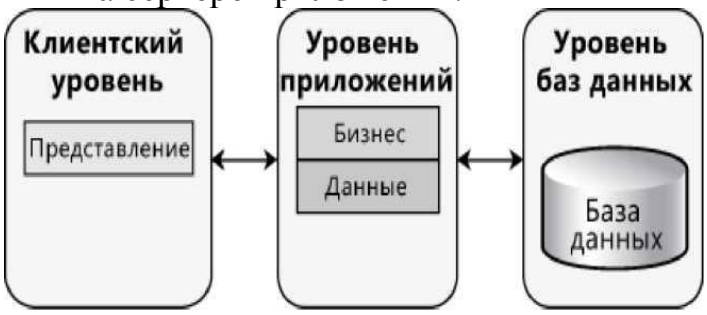

Рис. 2.12 Насыщенный клиент, бизнес-слой которого размещается на уровне приложений

Шаблоны развертывания для обеспечения наилучшей производительности

Шаблоны развертывания для обеспечения наилучшей производительности представляют проверенные проектные решения типовых проблем, связанных с производительностью.

Если требуется обеспечить высокую производительность при развертывании, можно использовать вертикальное или горизонтальное масштабирование.

Вертикальное масштабирование подразумевает улучшение оборудования, на котором уже выполняется приложение.

Горизонтальное масштабирование подразумевает размещение приложения на множестве физических серверов для распределения нагрузки. По сути, горизонтальное масштабирование – это реализация стратегии балансировки нагрузки. В этой связи часто применяются такие понятия как кластер с балансировкой нагрузки или, для Веб-серверов, Веб-ферма.

Кластер с балансировкой нагрузки

Сервис или приложение могут быть установлены на нескольких серверах, конфигурированных для разделения рабочей нагрузки, как показано на рис. 2.13. Такой тип конфигурации называется кластер с балансировкой нагрузки.

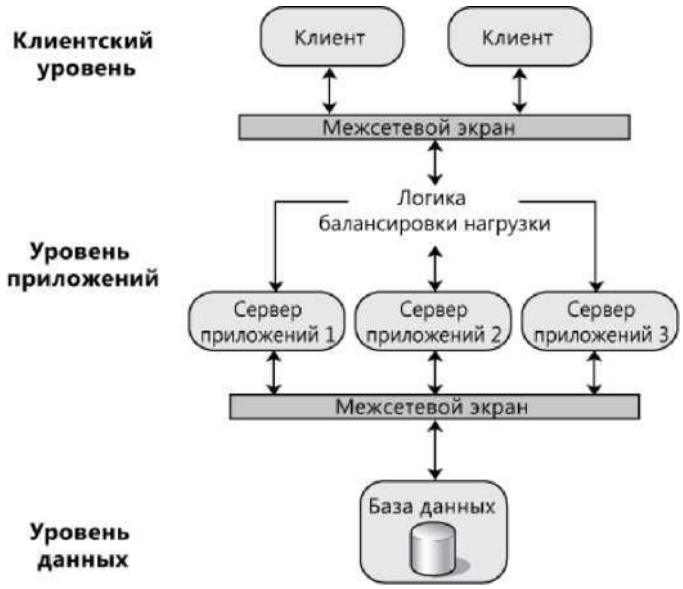

Рис. 2.13 Кластер с балансировкой нагрузки

Балансировка нагрузки позволяет масштабировать производительность серверных программ, таких как Веб-сервер, через распределение запросов клиента между множеством серверов.

Технологии балансировки нагрузки, которые обычно называют подсистемами балансировки нагрузки, принимают поступающие запросы и в случае необходимости перенаправляют их на определенный хост. Хосты с балансировкой нагрузки одновременно отвечают на запросы разных клиентов, даже на множество запросов одного клиента. Например, Веб-браузер может получать разные изображения для одной Веб-страницы с разных хостов кластера. Это позволяет распределять нагрузку, ускоряет обработку и сокращает время ответа.

Технологии маршрутизации позволяют выявлять вышедшие из строя серверы и удалять их из списка маршрутизации, чтобы максимально сократить последствия сбоя. В простых сценариях маршрутизация может быть реализована на основе механизмов циклического обслуживания, когда DNS-сервер последовательно перебирает все адреса имеющихся серверов. Рис. 2.14 иллюстрирует простую Веб-ферму (кластер Веб-серверов с балансировкой нагрузки), в которой на каждом сервере размещены все уровни приложения, кроме хранилища данных.

Кластеры с балансировкой нагрузки обеспечивают большую масштабируемость и эффективность, если им не приходится отслеживать и сохранять данные между каждым клиентским запросом, иначе говоря, если они реализованы без сохранения состояния. Если требуется отслеживать состояние, вероятно, лучше использовать методики привязки клиента к конкретному серверу и сеансы.

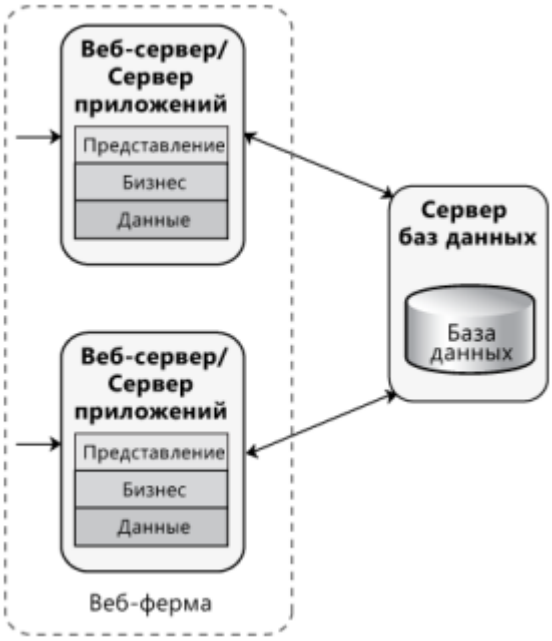

Рис. 2.14 Простая Веб-ферма

Привязка к конкретному серверу и сеансы пользователей

Приложения могут полагаться на сохранение состояния сеанса между запросами от одного клиента. Веб-серверу, например, может понадобиться отслеживать пользовательские данные между запросами. Веб-ферма может быть конфигурирована так, что все запросы от одного пользователя будут направляться на один и тот же сервер – это называется привязкой к конкретному серверу (affinity) – для сохранения состояния, когда эти данные сохраняются в памяти на Веб-сервере. Однако для повышения доступности и надежности на Веб-ферме следует использовать отдельное хранилище для данных о состоянии, это устранит требование привязки к конкретному серверу. Internet Information Services (IIS) 6.0 и последующие его версии можно настроить на работу в режиме Веб-сад, это обеспечит корректную обработку данных о состоянии сеанса в приложении во время его разработки.

В ASP.NET там, где не реализовывается привязка к конкретному серверу, все Веб-серверы должны быть настроены на использование одного ключа и метода шифрования для шифрования ViewState. Там, где система поддерживает эту возможность, для сеансов, использующих шифрование по протоколу Secure Sockets Layer (SSL), должна быть включена привязка к конкретному серверу, либо для SSL-запросов должен применяться отдельный кластер.

Фермы приложений

Бизнес-слой и слой доступа к данным, если они располагаются отдельно от уровня представления, на другом физическом уровне, могут масштабироваться аналогично Веб-серверам и Веб-фермам. Для этого используется ферма приложений. Запросы с уровня представления распределяются между серверами фермы так, чтобы каждый из них был загружен примерно одинаково. Компоненты бизнес-уровня и компоненты уровня доступа к данным можно распределить по разным фермам приложений в зависимости от требований каждого уровня, а также предполагаемой нагрузки и количества пользователей.

Шаблоны развертывания для обеспечения надежности

Шаблоны развертывания для обеспечения надежности представляют проверенные решения типовых проблем, связанных с надежностью. Наиболее распространенным подходом для повышения надежности развертывания является использование отказоустойчивого кластера, который гарантирует доступность приложения даже в случае сбоя одного из серверов.

Отказоустойчивый кластер

Отказоустойчивый кластер - это набор серверов, конфигурированных таким образом, что в случае отказа одного из них, другие серверы принимают на себя его нагрузку и продолжают обработку. На рис. 2.15 представлен отказоустойчивый кластер.

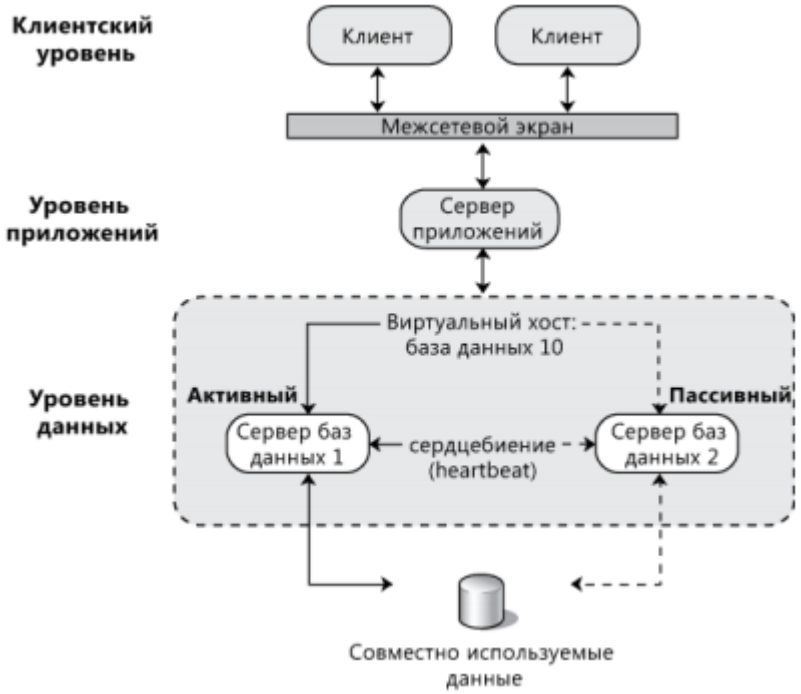

Рис. 2.15 Отказоустойчивый кластер

Установите приложение или сервис на множестве серверов, конфигурированных так, чтобы нагрузка вышедшего из строя сервера перераспределялась между оставшимися. Процесс перевода нагрузки на другой сервер в случае сбоя сервера называют обработкой отказа (failover). Каждый сервер в кластере имеет, по крайней мере, один резервный сервер

Шаблоны обеспечения безопасности

Шаблоны обеспечения безопасности представляют проверенные решения типовых проблем, связанных с безопасностью.

Хорошим подходом для передачи контекста исходной вызывающей стороны на нижние уровни или компоненты приложения является олицетворение/делегирование.

Подход с применением доверенной подсистемы подойдет, если требуется реализовывать аутентификацию и авторизацию в вышестоящих компонентах и выполнять доступ к нижестоящему ресурсу с использованием единственного доверенного удостоверения.

Олицетворение /делегирование

В модели олицетворения/делегирования доступ к ресурсам и типам операций (таким как чтение, запись и удаление) контролируется посредством списков управления доступом Windows (Windows Access Control Lists, ACLs) или эквивалентных средств безопасности целевых ресурсов (таких как таблицы и процедуры SQL Server). Доступ к ресурсам пользователями осуществляется с применением их исходных удостоверений через олицетворение, как показано на рис. 2.16. Нельзя забывать, что в случае применения этого подхода может возникнуть требование о наличии доменной учетной записи, что делает его непривлекательным для некоторых сценариев.

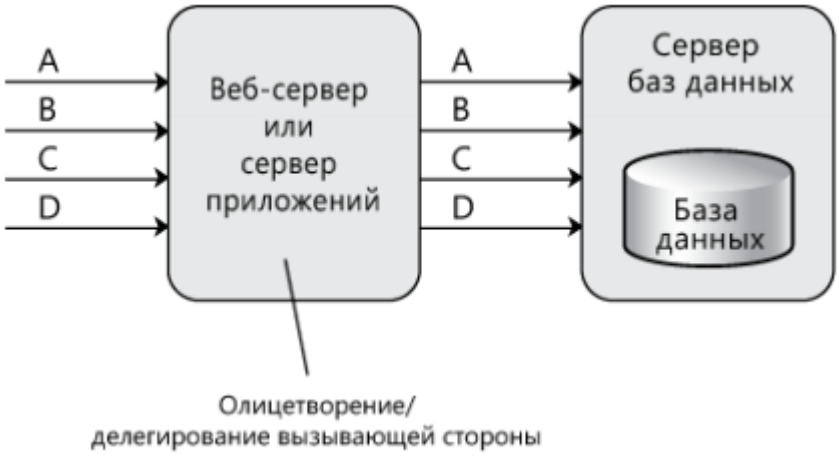

Рис. 2.16 Модель авторизации с олицетворением/делегированием

Доверенная подсистема

В модели доверенная подсистема (или доверенный сервер) пользователи логически сгруппированы соответственно описанным приложением ролям.

Участники определенной роли обладают одинаковыми привилегиями в приложении. Авторизация доступа к операциям (обычно осуществляющегося в виде вызовов методов) происходит на основании принадлежности вызывающей стороны к той или иной роли. При таком основанном на ролях (или операциях) подходе к обеспечению безопасности авторизация доступа к операциям (не сетевым ресурсам) выполняется на основании ролевой принадлежности вызывающей стороны. Роли, определяемые на этапе проектирования приложения, используются как логические контейнеры для группировки

пользователей с одинаковыми привилегиями или возможностями в приложении. Сервис промежуточного уровня использует фиксированное удостоверение для доступа к нижестоящим сервисам и ресурсам, как показано на рис. 2.16.

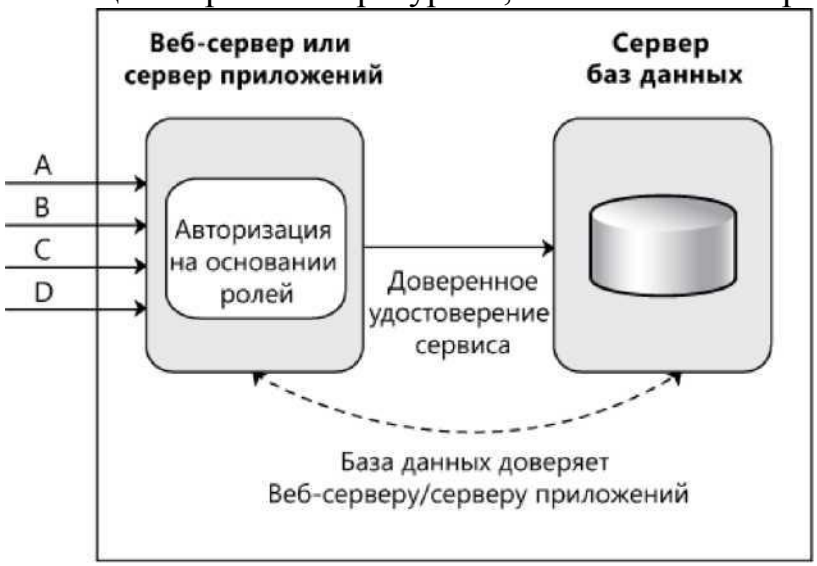

Рис. 2.16 Модель доверенная подсистема

Модель с применением множества доверенных удостоверений сервиса

В некоторых ситуациях может потребоваться более одного доверенного удостоверения, например, при наличии двух групп пользователей, одна из которых должна быть авторизована на осуществление операций чтения/записи, а другая - только операций чтения.

Использование множества доверенных удостоверений сервиса обеспечивает возможность более детального контроля доступа к ресурсам и аудита без особого влияния на масштабируемость. На рис. 2.17 показана модель с применением множества доверенных удостоверений сервиса.

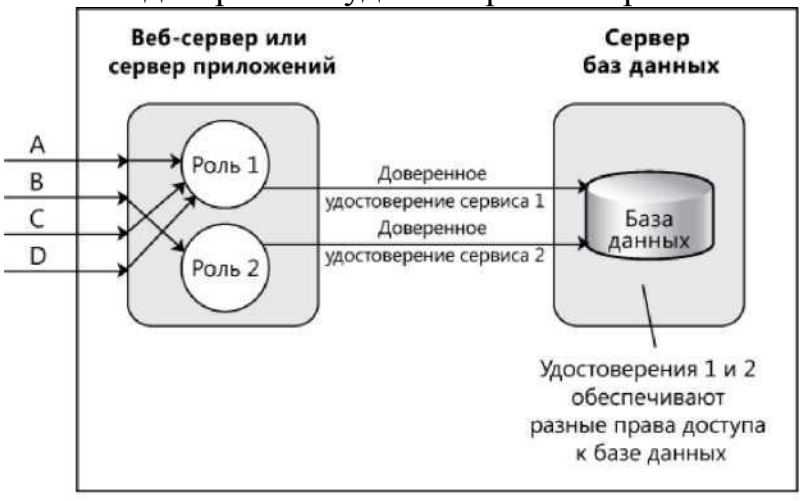

Рис. 2.17 Модель с применением множества доверенных удостоверений сервиса

Вертикальное и горизонтальное масштабирование

Подход к реализации масштабирования является критически важным аспектом проектирования. Независимо от того, планируется ли выполнять

горизонтальное масштабирование решения с помощью кластера с балансировкой нагрузки или секционированной базы данных, дизайн должен обеспечивать поддержку выбранной опции.

Существует два основных типа масштабирования: вертикальное (большой блок) и горизонтальное (больше блоков).

При вертикальном масштабировании поддержка повышенной нагрузки обеспечивается через введение в существующие серверы дополнительного оборудования, такого как процессоры, оперативная память и сетевые интерфейсные платы (network interface cards, NIC). Такой простой вариант не добавляет затрат на обслуживание и поддержку, но может быть экономически выгодным лишь до определенного момента. Однако всегда сохраняется вероятность сбоя, что является риском. Кроме того, введение дополнительного оборудования в существующие серверы обеспечивает желаемые результаты не бесконечно, и получение последних 10% расчетной производительности путем наращивания мощностей одного компьютера может быть очень дорогим удовольствием.

Эффективного вертикального масштабирования приложения можно добиться лишь при условии соответствующего вертикального масштабирования базовой инфраструктуры, среды выполнения и архитектуры компьютера. Продумайте, какие ресурсы ограничивают производительность приложения. Например, если это связано с нехваткой памяти или низкой пропускной способностью сети, добавление процессоров ничего не даст.

При горизонтальном масштабировании добавляется больше серверов и используются решения с балансировкой нагрузки и кластеризацией. Кроме возможности обработки большей нагрузки, горизонтальное масштабирование смягчает последствия сбоев оборудования. Если один из серверов выходит из строя, другие серверы кластера берут на себя его нагрузку. Например, уровень представления и бизнес-уровень приложения могут размещаться на нескольких Веб-серверах с балансировкой нагрузки, образующих Веб-ферму. Или можно физически отделить бизнес-логику приложения и использовать для нее отдельный средний уровень с балансировкой нагрузки, но при этом размещать уровень представления на внешнем уровне с балансировкой нагрузки. Если приложение имеет ограничения по вводу/выводу и должно поддерживать очень большую базу данных, ее можно распределить по нескольким серверам баз данных. Как правило, способность приложения масштабироваться горизонтально больше зависит от его архитектуры, чем от базовой инфраструктуры.

3. Архитектура основных типов приложений

В данном разделе рассматриваются возможности, свойства, преимущества и недостатки всех базовых типов приложений, таких как Веб-приложение, мобильное приложение, насыщенный клиент, сервисы и RIA.

3.1 Типы приложений [9]

Выдвигаемые клиентом требования, ограничения технологии и планируемый тип взаимодействия с пользователем определяет используемый тип приложения. Например, необходимо заранее продумать, будут ли обслуживаемые клиенты иметь доступ к постоянному сетевому соединению, должно ли предоставляться в Веб-браузере насыщенное медиа-содержимое анонимным пользователям или приложение будет использоваться преимущественно небольшим числом пользователей корпоративной внутренней сети.

Рассмотрим основные базовые типы приложений:

– Мобильные приложения.

Приложения этого типа могут разрабатываться как тонкий клиент или насыщенное клиентское мобильные приложение. Насыщенные клиентские мобильные приложения могут поддерживать сценарии без постоянного подключения или без подключения вообще. Веб-приложения или тонкие клиентские приложения поддерживают только сценарии с подключением. Ограничением при разработке мобильных приложений могут быть устройства, на которых их предполагается выполнять.

– Насыщенные клиентские приложения.

Приложения этого типа обычно разрабатываются как самодостаточные приложения с графическим пользовательским интерфейсом, который обеспечивает отображение данных с помощью набора элементов управления. Насыщенные клиентские приложения могут поддерживать сценарии без подключения или без постоянного подключения, если должны выполнять доступ к удаленным данным или функциональности.

– Насыщенные Интернет-приложения.

Приложения этого типа могут поддерживать множество платформ и браузеров. Насыщенные Интернет-приложения выполняются в изолированной программной среде браузера, которая ограничивает доступ к некоторым возможностям клиента.

– Сервисные приложения.

Сервисы предоставляют бизнес-функциональность для совместного использования и позволяют клиентам доступ к ней из локальной или удаленной системы. Вызов операций сервиса осуществляется с помощью сообщений, соответствующих XML-схемам и передаваемых по транспортным каналам. Целью данного типа приложений является обеспечение слабой связанности между клиентом и сервером.

– Веб-приложения.

Приложения этого типа, как правило, поддерживают сценарии с постоянным подключением и различные браузеры, выполняющиеся в разнообразнейших операционных системах и на разных платформах.

---

[9] Лабораторные работы №№ 1- 6

Существует множество других более специализированных типов приложений. Как правило, эти типы являются специализациями или сочетаниями базовых типов, перечисленных в данном списке.

В таблице 3.1 перечислены преимущества и недостатки общих архетипов приложений.

Таблица 3.1

Тип приложения	**Преимущества**	**Недостатки**
Мобильные приложения	– поддержка портативных устройств – доступность и простота использования для мобильных пользователей; – поддержка сценариев без подключения и сценариев без постоянного подключения.	– ограниченные возможности ввода и навигации. – ограниченная область отображения экрана.
Насыщенные клиентские приложения	– возможность использования ресурсов клиента. – лучшее время отклика, насыщенная функциональность UI и улучшенное взаимодействие с пользователем. – очень динамичное взаимодействие с коротким временем отклика. – поддержка сценариев без подключения и сценариев без постоянного подключения.	– сложность развертывания; – сложности обеспечения совместимости версий – зависимость от платформы.
Насыщенные Интернет-приложения (RIA)	– такие же насыщенные возможности пользовательского интерфейса, как и для насыщенных клиентов. – поддержка насыщенных и потоковых мультимедиа и графики. – простота развертывания с возможностями	– больший объем памяти, занимаемый на клиенте, по сравнению с веб-приложением. – ограниченное использование ресурсов клиента по сравнению с насыщенным клиентским приложением. – необходимость развертывания на клиенте

Тип приложения	Преимущества	Недостатки
	распределения (насыщенными) такими же, как и для веб-клиентов. – простота обновления и смены версий. – поддержка различных платформ и браузеров.	подходящей среды выполнения.
Сервисные приложения	– слабо связанное взаимодействие между клиентом и сервером. – могут использоваться различными и невзаимосвязанными приложениями. – поддержка для обеспечения возможности взаимодействия.	– отсутствие поддержки UI. – зависимость от возможности сетевого подключения.
Веб-приложения	– широко доступный и основанный на стандартах UI, поддерживаемый на многих платформах. – простота развертывания и внесения изменений.	– необходимость устойчивого сетевого подключения. – сложно обеспечить насыщенный пользовательский интерфейс.

Каждый тип приложения может быть реализован с использованием одной или более технологий.

Выбор технологии определяется сценариями и ограничениями технологий, а также возможностями и опытом группы разработки.

Мобильное приложение

Мобильное приложение, как правило, структурируется как многослойное приложение, включающее слой пользовательского интерфейса (представления), бизнес-слой и слой доступа к данным (см. рис. 3.1).

Мобильное приложение может быть тонким Веб-клиентом или насыщенным клиентом.

Если создается насыщенный клиент, бизнес-слой и слой доступа к данным, скорее всего, будут располагаться на самом устройстве.

Для тонкого клиента бизнес-слой и слой доступа к данным будут располагаться на сервере.

Мобильные приложения обычно реализуют поддержку сценариев без подключения через использование локально кэшированных данных, синхронизация которых выполняется при установлении подключения. Они также могут использовать сервисы, предоставляемые другими приложениями,

включая размещаемые сервисы типа S+S (ПО + сервисы) и Веб-сервисы. Часто мобильному клиентскому приложению предоставляется управляемая синхронизация с источником данных и доступ к другим сервисам через специальную серверную инфраструктуру.

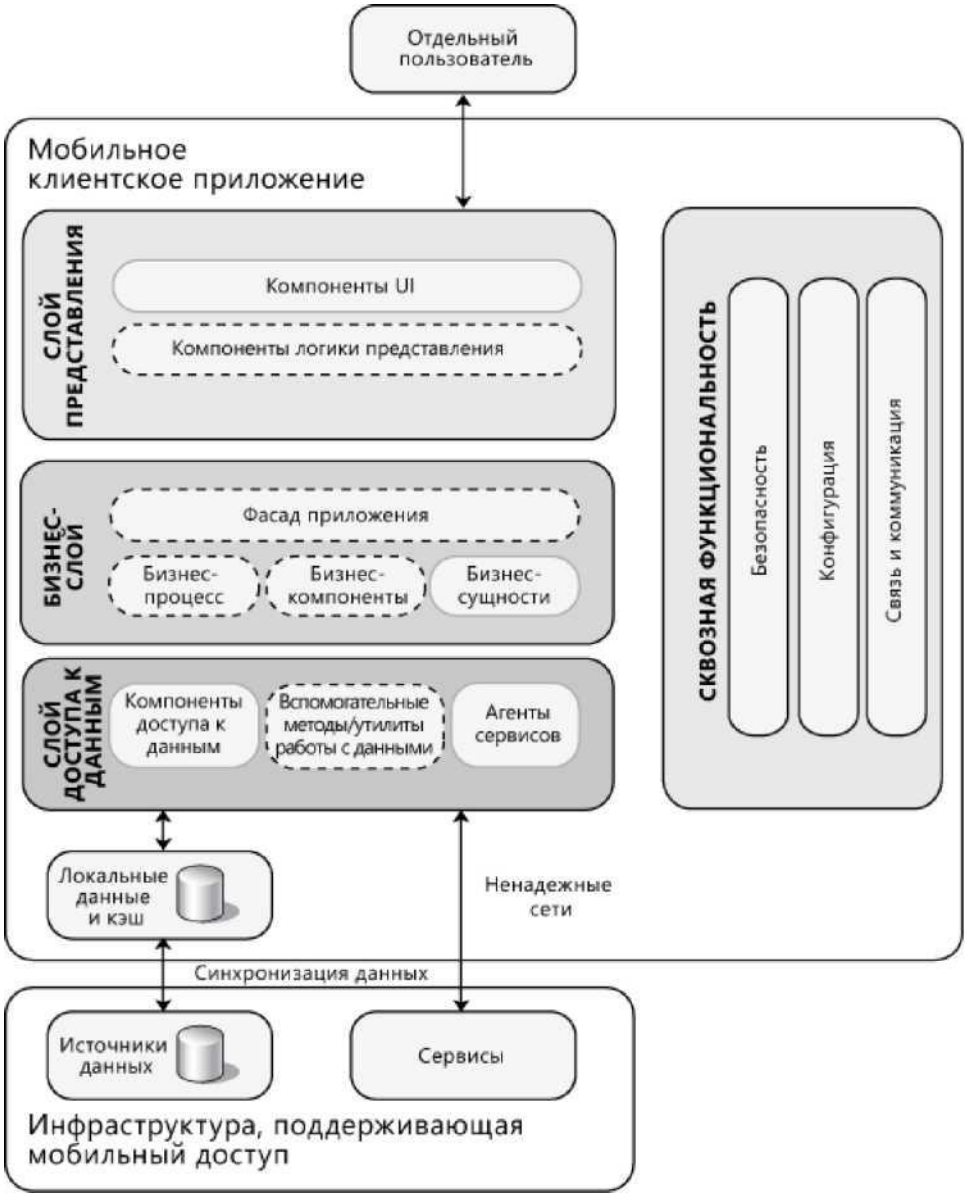

Рис. 3.1 Типовая структура мобильного приложения

Используйте мобильные приложения, если:

- пользователи зависят от портативных устройств;
- приложение поддерживает простой UI, подходящий для использования на небольшом экране;
- приложение должно поддерживать сценарии без подключения или без постоянного подключения, в этом случае более подходящим будет мобильное насыщенное клиентское приложение;
- приложение должно быть независимым от устройств и может зависеть от возможности сетевого подключения, в этом случае более подходящим будет мобильное веб-приложение.

Насыщенное клиентское приложение

Насыщенные клиентские пользовательские интерфейсы могут обеспечить интерактивное, насыщенное взаимодействие с пользователем с минимальным временем отклика для приложений, которые должны выполняться как самодостаточное приложение, в сценариях с подключением, без постоянного подключения и без подключения.

Как правило, насыщенное клиентское приложение структурировано как многослойное приложение, включающее слой пользовательского интерфейса (представления), бизнес-слой и слой доступа к данным (рис. 3.2)

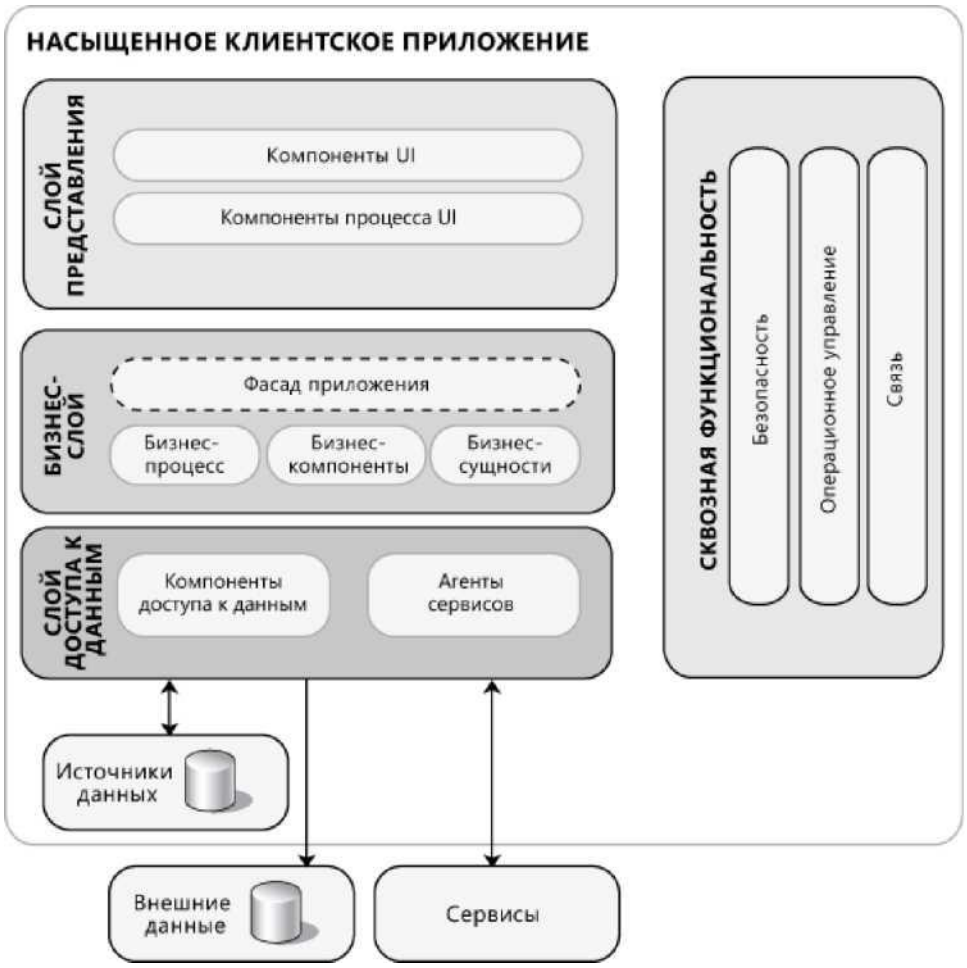

Рис. 3.2 Типовая структура насыщенного клиентского приложения

Насыщенное клиентское приложение может использовать данные с удаленного сервера, данные, хранящиеся локально, или данные из обоих типов источников. Также оно может потреблять сервисы, предоставляемые другими приложениями, включая размещаемые сервисы типа S+S и Веб-сервисы.

Используйте насыщенные клиентские приложения, если:

- приложение должно поддерживать сценарии без подключения или без постоянного подключения.
- приложение будет развертываться на клиентских ПК.
- приложение должно обеспечивать высокий уровень интерактивности и минимальное время отклика.

– UI приложения должен обеспечивать насыщенную функциональность и взаимодействие с пользователем, но без расширенных графических возможностей или возможностей воспроизведения мультимедиа RIA.
– приложение должно использовать ресурсы клиентского ПК.

Насыщенное Интернет-приложение

Насыщенное Интернет-приложение (RIA) выполняется в браузере в изолированной программной среде.

К преимуществам RIA, по сравнению с традиционными Веб-приложениями, относятся более насыщенный пользовательский интерфейс, улучшенное время отклика приложения и эффективность работы с сетью.

Как правило, RIA структурировано как многослойное приложение, включающее слой пользовательского интерфейса (представления), бизнес-слой и слой доступа к данным (рис. 3.3).

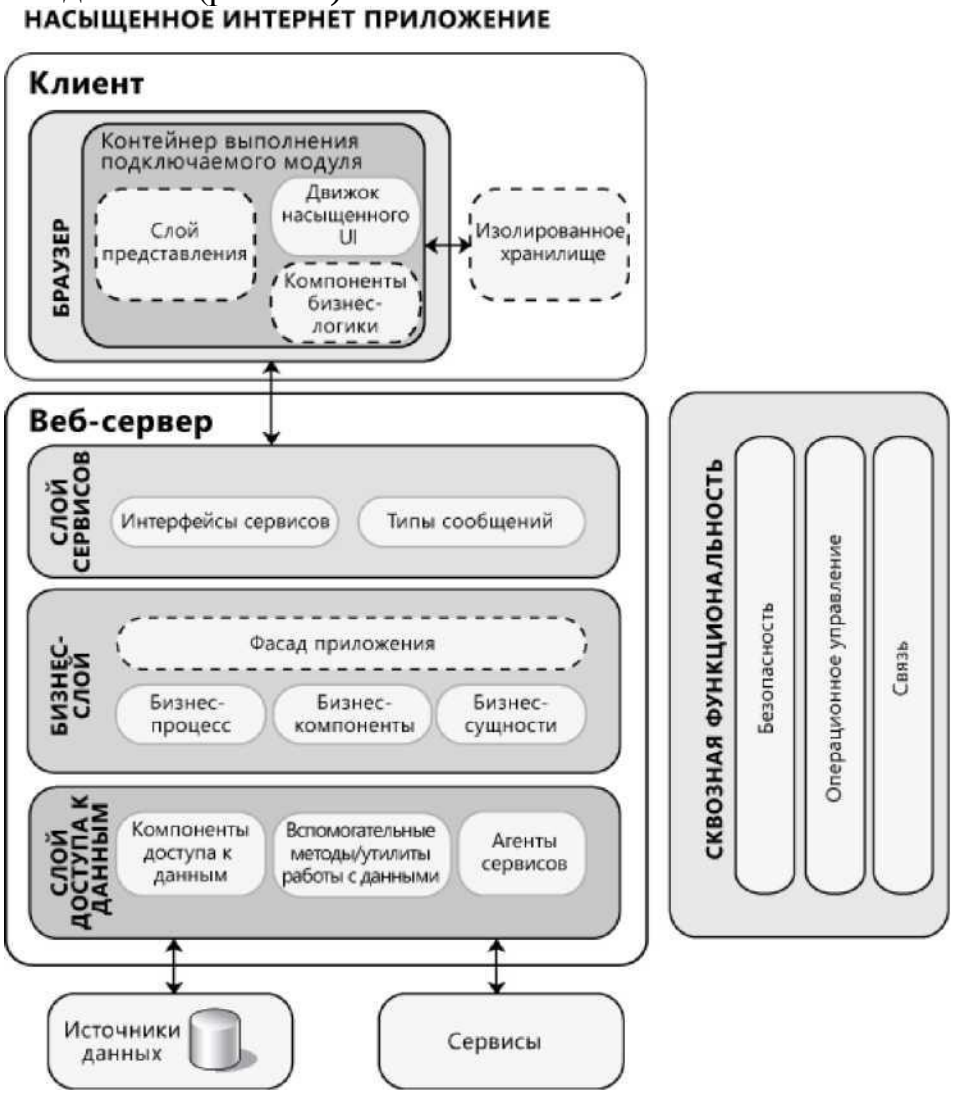

Рис. 3.3 Типовая структура насыщенного Интернет-приложения

Как правило, RIA-приложения зависят от подключаемого модуля на стороне клиента или размещаемой среды выполнения (такой как среда выполнения XAML или Silverlight). Подключаемый модуль взаимодействует с

удаленными хостами Веб-сервера, которые формируют код и данные, потребляемые клиентским подключаемым модулем или средой выполнения.

Используйте насыщенные Интернет-приложения, если:

- приложение должно поддерживать насыщенные мультимедиа и обеспечивать представление, насыщенное графическими элементами.
- приложение должно обеспечивать более насыщенный, интерактивный пользовательский интерфейс с меньшим временем отклика, чем Веб-приложения.
- использование приложением вычислительных мощностей клиента будет ограничено.
- использование приложением ресурсов клиента будет ограничено.
- требуется обеспечить простую модель развертывания в Веб.

Сервис

В контексте данной темы сервис - это открытый интерфейс, обеспечивающий доступ к единице функциональности.

Сервис, фактически, предоставляет программный сервис вызывающей стороне, которая потребляет этот сервис. Как правило, сервисное приложение, предоставляющее такие сервисы, структурировано как многослойное приложение, включающее слой сервисов, бизнес-слой и слой доступа к данным (рис. 3.4).

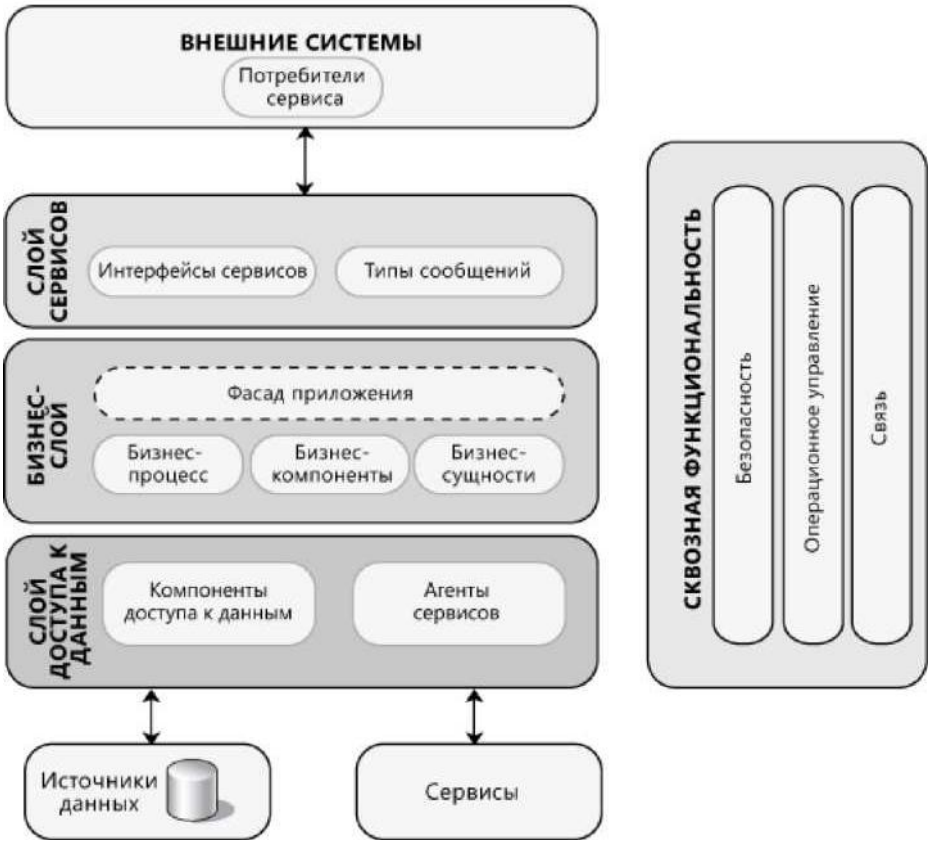

Рис. 3.4 Типовая структура сервисного приложения

Сервисы слабо связаны и могут сочетаться для обеспечения более сложной функциональности. Сервисы могут быть распределенными, доступ к сервису может осуществляться как удаленно, так и с компьютера, на котором сервис

выполняется. Также сервисы ориентируются на обмен сообщениями. Это означает, что интерфейсы описываются WSDL-документом, и операции вызываются с помощью построенных на XML-схемах сообщений, которые передаются по транспортному каналу. Кроме того, благодаря реализации взаимодействия через описание сообщения/интерфейса сервисы поддерживают гетерогенную среду. Если компоненты могут интерпретировать описание сообщения и интерфейса, они могут использовать сервис независимо от собственной базовой технологии.

Используйте сервисные приложения, если:

- приложение будет предоставлять функциональность, не требующую UI.
- необходимо обеспечить слабую связанность приложения с его клиентами.
- приложение должно совместно использоваться или потребляться другими внешними приложениями.
- приложение должно предоставлять функциональность, которая будет потребляться приложениями через интернет, интранет или на локальном компьютере.

Веб-приложение

В Веб-приложениях логика размещается на стороне сервера. Эта логика может состоять из множества отдельных слоев. Типовым примером является трехслойная архитектура, включающая слой представления, бизнес-слой и слой доступа к данным (рис. 3.5).

Слой представления обычно включает компоненты UI и логики представления; бизнес-слой – компоненты бизнес-логики, бизнес-процесса и бизнес-сущностей, а также иногда фасад; слой доступа к данным – компоненты доступа к данным и агенты сервисов.

Как правило, Веб-приложение осуществляет доступ к хранилищу данных на удаленном сервере базы данных. Также оно может потреблять сервисы, предоставляемые другими приложениями, включая размещаемые сервисы типа ПО + сервисы и Веб-сервисы.

Используйте Веб-приложения, если:

- приложению не требуется поддерживать насыщенный UI и мультимедиа, что предлагает насыщенное Интернет-приложение.
- требуется обеспечить простую модель развертывания в Веб.
- пользовательский интерфейс должен быть независимым от платформы.
- приложение должно быть доступным через Интернет.
- требуется максимально сократить зависимости на стороне клиента и потребление ресурсов, таких как дисковое пространство или вычислительные мощности процессора.

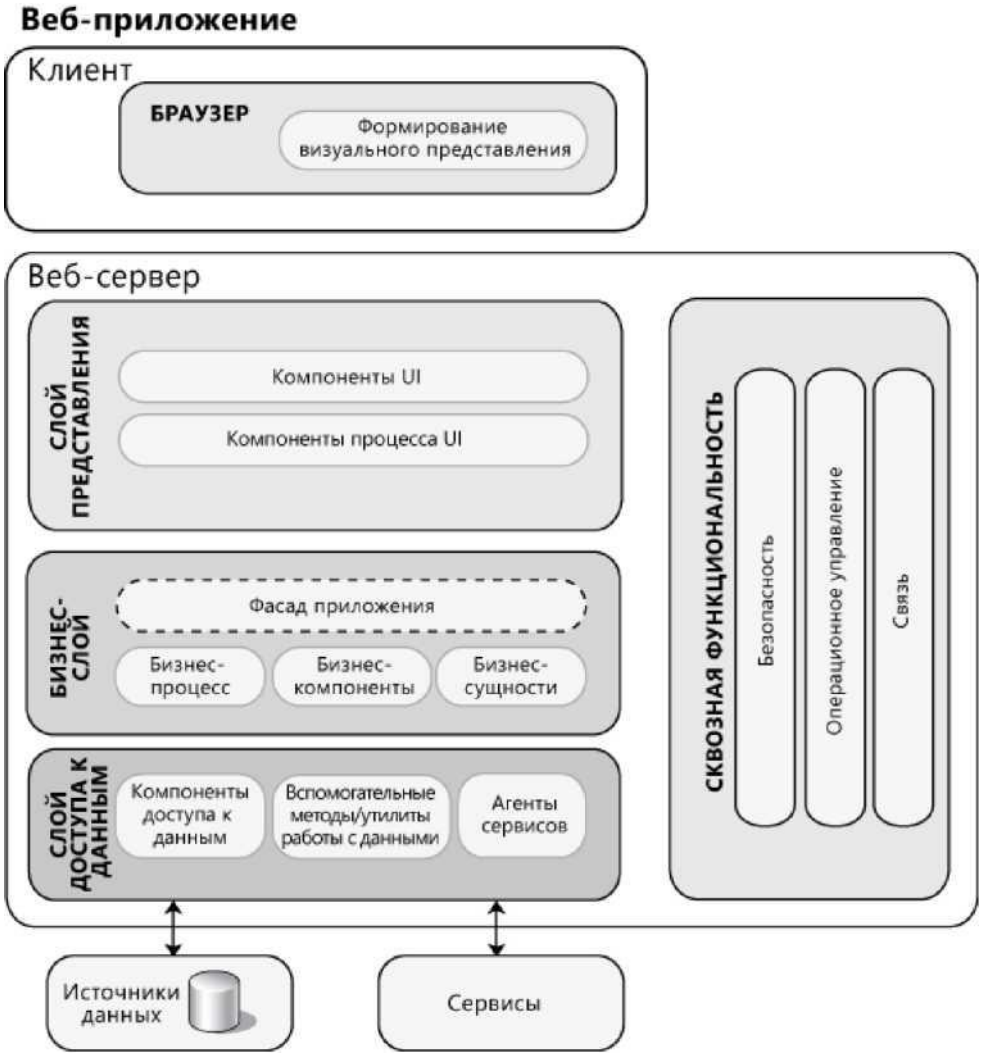

Рис. 3.5 Типовая структура Веб-приложения

3.2 Принципы проектирования и основные атрибуты Веб-приложения

Веб-приложение – это приложение, которое может использоваться пользователями через браузер или специализированный агент пользователя.

Браузер создает HTTP-запросы к определенным URL, которые сопоставляются с ресурсами на Веб-сервере. Сервер формирует визуальное представление HTML-страниц, которое может быть отображено браузером, и возвращает их клиенту.

Принципы проектирования

Основной целью архитектора ПО при проектировании Веб-приложения является максимальное упрощение дизайна через разделение задач на функциональные области, обеспечивая при этом безопасность и высокую производительность.

Данные рекомендации помогут выполнить все требования и создать условия для эффективной работы приложения в свойственных Веб-приложениям сценариях:

– Выполните логическое разделение функциональности приложения.

Используйте многослойную структуру для логического разделения приложения на слой представления, бизнес-слой и слой доступа к данным. Это

поможет создать удобный в обслуживании код и позволит отслеживать и оптимизировать производительность каждого слоя в отдельности. Четкое логическое разделение также обеспечивает более широкие возможности масштабирования приложения.

– Используйте абстракцию для реализации слабого связывания между слоями.

Этот подход можно реализовать путем определения интерфейсных компонентов, таких как фасад с общеизвестными входными и выходными параметрами, который преобразует запросы в формат, понятный компонентам слоя. Кроме того, с помощью интерфейсных типов или абстрактных базовых классов можно определить совместно используемую абстракцию, которая должна быть реализована интерфейсными компонентами.

– Определитесь с тем, как будет реализовано взаимодействие компонентов друг с другом.

Для этого необходимо понимать сценарии развертывания, которые должно поддерживать приложение. Выясните, должно ли поддерживаться взаимодействие через физические границы или границы процесса, либо все компоненты будут выполняться в рамках одного процесса.

– Используйте кэширование для сокращения количества сетевых вызовов и обращений к базе данных.

При проектировании Веб-приложения используйте такие техники, как кэширование и буферизация вывода, для сокращения сетевых вызовов между браузером и Веб-сервером и Веб-сервером и нижестоящими серверами. Правильно спроектированная стратегия кэширования, вероятно, единственный наиболее важный с точки зрения производительности аспект дизайна. ASP.NET предоставляет следующие возможности кэширования: кэширование вывода страницы, частичное кэширование страниц и Cache API. Используйте эти возможности при проектировании приложения.

– Используйте протоколирование и инструментирование.

Необходимо выполнять аудит и протоколирование действий в слоях и уровнях приложения. Журналы регистрации событий могут использоваться для выявления подозрительных действий, что часто обеспечивает раннее обнаружение атак на систему. Не забывайте, что могут возникнуть сложности с регистрацией проблем, возникающих в коде сценариев, выполняющихся в браузере.

– Продумайте аспекты аутентификации пользователей на границах доверия.

При проектировании приложения необходимо предусмотреть аутентификацию пользователей при пересечении границ доверия, например, при доступе удаленного бизнес-слоя со слоя представления.

– Не передавайте конфиденциальные данные по сети в виде открытого текста.

Если требуется передавать по сети конфиденциальные данные, такие как пароль или cookie аутентификации, используйте для этого шифрование и

подписи данных либо шифрование с использованием протокола Secure Sockets Layer (SSL).

- Проектируйте Веб-приложение для выполнения под менее привилегированной учетной записью.

Процесс должен иметь ограниченный доступ к файловой системе и другим ресурсам системы. Это позволит максимально сократить возможные негативные последствия на случай, если злоумышленник попытается взять процесс под свой контроль.

3.3 Проектирование насыщенных клиентских приложений

Пользовательские интерфейсы насыщенных клиентов могут обеспечивать высокую производительность, интерактивность и насыщенное взаимодействие с пользователем для приложений, которые должны работать как самодостаточные приложения и в сценариях с подключением, без постоянного подключения и без подключения.

Windows Forms, Windows Presentation Foundation (WPF) и Microsoft Office Business Application (OBA) – среды и инструменты разработки, позволяющие разработчикам создавать насыщенные клиентские приложения.

Эти технологии могут использоваться для создания как самодостаточных приложений, так и приложений, выполняющихся на клиентском компьютере, но взаимодействующих с сервисами, предоставляемыми другими уровнями (как логическими, так и физическими), и другими приложениями, предоставляющими операции, которые необходимы клиенту. К этим операциям относятся доступ к данным, извлечение данных, поиск, отправка данных в другие системы, резервное копирование и т.д. На рис. 3.2 показан общий вид архитектуры типового насыщенного клиента и показаны компоненты, обычно располагающиеся в каждом из слоев.

Типовое насыщенное клиентское приложения включает три слоя: слой представления, бизнес-слой и слой доступа к данным.

Слой представления, как правило, содержит компоненты UI и логики представления; бизнес-слой - компоненты бизнес-логики, бизнес-процесса и бизнес-сущностей; и слой доступа к данным - компоненты доступа к данным и агентов сервисов.

Насыщенные клиентские приложения могут быть довольно тонкими приложениями, состоящими, главным образом, из слоя представления, который с помощью сервисов выполняет доступ к удаленному бизнес-уровню, размещаемому на серверах. Примером такого приложения является приложение для ввода данных, передающее все данные на сервер для обработки и хранения.

И наоборот, насыщенные клиентские приложения могут быть очень сложными приложениями, которые осуществляют большую часть обработки самостоятельно и взаимодействуют с другими сервисами и хранилищами данных для получения или отправки данных. Примером такого приложения является ПО на базе Microsoft Excel®, которое выполняет сложные задачи локально, сохраняет состояние и данные локально и взаимодействует с удаленными серверами только для извлечения и обновления связанных данных.

Такие насыщенные клиенты имеют собственные бизнес-слои и слои доступа к данным. Рекомендации по проектированию бизнес-слоев и слоев доступа к данным в таких приложениях аналогичны рекомендациям для всех остальных приложений.

Принципы проектирования

Целью архитектора ПО при проектировании насыщенного клиентского приложения является правильный выбор технологии, и создание максимально простой структуры за счет распределения задач по разным функциональным областям. Дизайн должен отвечать предъявляемым к приложению требованиям по производительности, безопасности, возможности повторного использования и простоты обслуживания.

При проектировании насыщенных клиентских приложений руководствуйтесь следующими рекомендациями:

- Выбирайте соответствующую технологию, исходя из требований, предъявляемых к приложению.

К таким технологиям относятся Windows Forms, WPF, XAML Browser Applications (XBAP) и OBA.

- Разделяйте логику представления и реализацию интерфейса.

Используйте такие шаблоны проектирования, как Presentation Model и Supervising Presenter (или Supervising Controller), которые отделяют формирование визуального отображения UI от логики UI. Использование отдельных компонентов в приложении сокращает количество зависимостей, упрощает обслуживание и тестирование и создает лучшие условия для повторного использования.

- Выясните задачи представления и потоки представления.

Это поможет спроектировать все окна и шаги с использованием многооконного подхода или мастера.

- Спроектируйте подходящий и удобный интерфейс.

Для обеспечения максимальной доступности, удобства и простоты использования учтите такие аспекты, как компоновка, навигация, выбор элементов управления и локализация.

- Применяйте разделение функциональных областей во всех слоях.

Например, вынесите бизнес-правила и другие задачи, не связанные с представлением, в отдельный бизнес-слой, а код доступа к данным в отдельные компоненты, размещаемые в слое доступа к данным.

- Используйте существующую общую логику представления.

Библиотеки, содержащие шаблонные, обобщенные функции валидации на стороне клиента, и вспомогательные классы могут использоваться повторно многими приложениями.

- Обеспечивайте слабое связывание клиента с удаленными сервисами, которые он использует.

Применяйте интерфейс взаимодействия на основании сообщений для взаимодействия с сервисами, размещенными на других уровнях.

- Избегайте тесного связывания с объектами других слоев.

При взаимодействии с другими слоями приложения используйте абстракцию, предоставляемую общими описаниями интерфейсов, абстрактные базовые классы или обмен сообщениями. Например, реализация шаблонов Dependency Injection и Inversion of Control может обеспечить абстракцию, которая будет использоваться при взаимодействии между слоями.

– Сократите количество обращений к сети при доступе к удаленным уровням.

Применяйте слабо детализированные методы и по возможности выполняйте их асинхронно, чтобы избежать блокировки UI или его перехода в состояние «не отвечает».

3.4 Проектирование насыщенных Веб- приложений

Насыщенные Веб-приложения (Rich Internet Applications, RIA) поддерживают насыщенные графические элементы и сценарии с применением потокового мультимедиа, обеспечивая при этом преимущества развертывания и удобства обслуживания, присущие Веб-приложению.

RIA могут выполняться в подключаемом модуле браузера, таком как Microsoft® Silverlight®, в отличие от расширений, использующих код браузера, таких как Asynchronous JavaScript и XML (AJAX).

Типовая реализация RIA использует Веб-инфраструктуру в сочетании с клиентским приложением, отвечающим за обработку представления. Подключаемый модуль обеспечивает библиотечные процедуры для поддержки насыщенной графики, а также контейнер, в целях безопасности ограничивающий доступ к локальным ресурсам.

RIA могут выполнять более расширенный и сложный код на стороне клиента, чем обычное Веб-приложение, обеспечивая тем самым возможность сократить нагрузку на Веб-сервер. На рис. 3.3 представлена типовая структура реализации RIA.

Типовое насыщенное Интернет-приложение включает три слоя: слой представления, бизнес-слой и слой доступа к данным.

Слой представления, как правило, содержит компоненты UI и логики представления; бизнес-слой – компоненты бизнес-логики, бизнес-процесса и бизнес-сущностей; и слой доступа к данным – компоненты доступа к данным и агентов сервисов.

В RIA часть бизнес-процессов и даже код доступа к данным часто переносятся на клиент. Поэтому клиентская часть, в зависимости от сценария, может включать некоторую или всю функциональность бизнес-слоя и слоя доступа к данным. На рис. 3.3 показано, как некоторые бизнес-процессы обычно реализовываются на клиенте.

RIA могут быть как тонкими интерфейсами для серверных бизнес-сервисов, так и сложными приложениями, которые самостоятельно выполняют большую часть процессов и взаимодействуют с сервисами на сервере только для получения или отправки данных. Следовательно, дизайн и реализация RIA могут быть очень разнообразными. Тем не менее, существует ряд общих подходов, обеспечивающих создание хорошей архитектуры слоя

представления и его взаимодействия с сервисами на сервере. Многие из них основываются на общеизвестных шаблонах проектирования, которые способствуют построению приложения из отдельных компонентов. Это обеспечивает сокращение зависимостей, упрощение обслуживания и тестирования, а также более широкие возможности повторного использования.

Принципы проектирования

Представленные далее рекомендации касаются нескольких аспектов, которые должны быть учтены при проектировании RIA, и помогут обеспечить эффективную работу приложения в общих для RIA сценариях и соответствие выдвигаемым требованиям:

- Выбирайте RIA, исходя из предполагаемой аудитории, насыщенного интерфейса и простоты развертывания.

Создавайте RIA, если целевая аудитория использует браузер, поддерживающий RIA. Если часть целевой аудитории работает с браузером, не поддерживающим RIA, рассмотрите возможность ограничения номенклатуры доступных для выбора браузеров только поддерживаемой версией. Если влиять на выбор браузера нет возможности, подумайте, достаточно ли велика неохватываемая аудитория, чтобы выбирать другой тип приложения, такой как Веб-приложение, использующее AJAX. Если клиенты имеют надежное сетевое соединение, развертывать и обслуживать RIA настолько же просто, как и Веб-приложение. Реализации RIA прекрасно подходят для Веб-сценариев, в которых требования, предъявляемые к визуализации, превышают возможности, обеспечиваемые базовым HTML. RIA обеспечивают более устойчивое поведение и требуют менее развернутого тестирования в поддерживаемых браузерах по сравнению с Веб-приложениями, использующими расширенные функции и код. Реализации RIA также идеально подходят для приложений, работающих с потоковыми мультимедиа. Они не так хороши для сверхсложных многостраничных UI.

- Используйте Веб-инфраструктуру посредством сервисов.

Реализациям RIA необходима такая же инфраструктура, что и Веб-приложениям. Как правило, вся обработка RIA и также взаимодействие с другими сетевыми сервисами (для сохранения данных в базе данных, например) выполняются на клиенте.

- Используйте вычислительные мощности клиента.

RIA выполняются на клиентском компьютере и могут использовать все доступные на нем вычислительные мощности. По возможности максимально перенесите функциональность на клиент, это позволит улучшить взаимодействие с пользователем. Однако особо важные бизнес-правила должны выполняться на сервере, поскольку проверка на клиенте может быть сфальсифицирована.

- Обеспечивайте выполнение в безопасной программной среде браузера.

Реализации RIA обладают более высокой безопасностью по умолчанию, поэтому не имеют доступа ко всем ресурсам компьютера, таким как камеры и

аппаратное ускорение видео. Доступ к локальной файловой системе ограничен. Локальное хранилище доступно, но с максимальными ограничениями.

– Определитесь с тем, насколько сложным будет UI.

Продумайте, насколько сложным будет создаваемый UI. Реализации RIA лучше работают при использовании одного окна для всех операций. Возможно применение множества окон, но это требует написания дополнительного кода и обработки переходов между окнами. Пользователи должны иметь возможность без труда перемещаться или останавливаться и возвращаться к соответствующему окну без повторного запуска всего процесса. Для многостраничных UI применяйте методы внешнего связывания. Также управляйте унифицированным указателем ресурса (Uniform Resource Locator, URL), историей и кнопками назад/вперед браузера во избежание сложностей при переходе пользователей между окнами.

– Используйте сценарии для повышения производительности или сокращения времени отклика приложения.

Проанализируйте общие сценарии приложений и примите решение о разделении и загрузке компонентов приложения, а также о кэшировании данных или переносе бизнес-логики на клиент. Чтобы сократить время загрузки и запуска приложения, разбивайте функциональность на отдельные загружаемые компоненты.

– Предусмотрите ситуацию отсутствия установленного подключаемого модуля.

Реализации RIA требуют наличия подключаемого модуля браузера, поэтому необходимо предусмотреть сценарий установки подключаемого модуля как часть выполнения приложения. Учтите то, имеют ли ваши клиенты доступ к подключаемому модулю, разрешение на его установку, и захотят ли они его устанавливать. Подумайте над тем, насколько детально будете контролировать процесс установки. Предусмотрите ситуацию, когда пользователи не могут установить подключаемый модуль, с выводом на экран информативного сообщения об ошибке или предоставлением альтернативного пользовательского Веб-интерфейса.

Кроме приведенных выше рекомендаций по реализаций RIA, руководствуйтесь и более общими советами по проектированию насыщенных клиентских приложений (включая мобильные насыщенные клиенты). К общим рекомендациям относятся отделение логики представления от реализации интерфейса, определение задач представления и потоков представления, разделение бизнес-правил и других задач, не связанных с интерфейсом, повторное использование общей логики представления, слабое связывание клиента с используемыми им удаленными сервисами, недопущение тесного связывания с объектами других слоев и сокращение числа обращений к сети при доступе к удаленным слоям.

3.5 Проектирование мобильных приложений

Как правило, обычное мобильное приложение структурировано как многослойное приложение, состоящее из слоя представления, бизнес-слоя и слоя

доступа к данным. При разработке мобильного приложения можно создавать тонкий Веб-клиент или насыщенный клиент.

Для насыщенного клиента бизнес-слой и слой сервисов данных, скорее всего, будут располагаться на самом устройстве. Для тонкого клиента все слои будут размещены на сервере. Рис. 3.1 иллюстрирует типовую архитектуру насыщенного клиентского мобильного приложения, компоненты которого сгруппированы по функциональным областям.

Как правило, в слое представления мобильного приложения располагаются компоненты пользовательского интерфейса и также, возможно, компоненты логики представления.

Бизнес-слой, если таковой имеется, обычно включает компоненты бизнес-логики, все компоненты бизнес-процесса и бизнес-сущностей, необходимые приложению, и фасад, если он используется.

В слое доступа к данным находятся компоненты доступа к данным и агентов сервисов.

Чтобы сократить объем занимаемой приложением памяти устройства, мобильные приложения обычно используют менее жесткие подходы к разделению на слои и меньшее число компонентов.

Принципы проектирования

Приведенные далее рекомендации по проектированию касаются различных аспектов, которые должны быть учтены при проектировании мобильного приложения. Они помогут обеспечить эффективную работу приложения в типовых сценариях мобильных приложений и соответствие всем выдвигаемым требованиям:

- Определитесь, создается ли насыщенный клиент, тонкий Веб-клиент или насыщенное Интернет-приложение (RIA).

Если создаваемое приложение требует локальной обработки и должно работать в сценарии без постоянного подключения, проектируйте насыщенный клиент. Насыщенное клиентское приложение будет сложнее устанавливать и обслуживать. Если приложение может зависеть от обработки на сервере и будет иметь устойчивое постоянное подключение, создавайте тонкий клиент. Если приложению необходим насыщенный UI, оно имеет только ограниченный доступ к локальным ресурсам и должно быть портируемым на другие платформы, используйте RIA-клиент.

- Определите, какие типы устройств будут поддерживаться.

При выборе поддерживаемых типов устройств обратите внимание на размер и разрешение экрана, характеристики производительности ЦП, объем памяти и хранилища, а также доступность среды разработки. Кроме того, учтите требования пользователей и ограничения организации. Может потребоваться специальное оборудование, такое как глобальная система определения местоположения (global positioning system, GPS) или камера, что может иметь влияние не только на тип приложения, но также на выбор устройства.

- В случае необходимости учтите сценарии без постоянного подключения и с ограниченной полосой пропускания.

Для автономных мобильных устройств нет необходимости учитывать аспекты подключения. Если мобильному приложению требуется возможность сетевого подключения, оно должно обрабатывать сценарии с неустойчивым подключением или без такового. В таких случаях крайне важно спроектировать механизмы кэширования, управления состоянием и доступа к данным в условиях неустойчивого подключения и пакетное взаимодействие, когда подключение доступно. Выбирайте оборудование и программные протоколы, руководствуясь скоростью, энергопотреблением и глубиной детализации, а не только исходя из простоты разработки.

- Проектируйте UI, подходящий для мобильных устройств, учитывая ограничения платформы.

Чтобы мобильные приложения могли работать в условиях ограничений, налагаемых мобильными устройствами, они должны иметь более простую архитектуру, простой UI, для них должны приниматься особые проектные решения. Учитывайте эти ограничения и не пытайтесь использовать архитектуру или UI настольного или Веб-приложения, а проектируйте для конкретного устройства. Основными ограничениями являются память, время работы батареи, способность адаптироваться к разным размерам или ориентации экрана, безопасность и пропускная способность сети.

- Создавайте многослойную архитектуру, подходящую для мобильных устройств, которая повышает возможности повторного использования и удобство обслуживания.

В зависимости от типа приложения все слои могут располагаться на самом устройстве. Используйте концепцию многослойности, чтобы обеспечить разделение функциональных областей и повысить возможности повторного использования и удобство обслуживания мобильного приложения. Тем не менее, стремитесь к тому, чтобы приложение занимало минимальный объем памяти на устройстве, упрощая дизайн по сравнению с настольным или Веб-приложением.

- Учитывайте ограничения, налагаемые ресурсами устройства, такие как время работы батареи, объем памяти и частота процессора.

Каждое проектное решение должно учитывать ограничения ЦП, памяти, емкость хранилища и время работы батареи мобильного устройства. Как правило, первостепенным ограничивающим фактором в мобильных устройствах является время работы батареи. Подсветка, чтение и запись в память, беспроводное соединение, специальное оборудование и частота процессора - все эти аспекты оказывают влияние на общее энергопотребление. При недостаточном объеме доступной памяти операционная система Windows Mobile может попросить закрыть приложение или пожертвовать кэшированными данными, что приводит к снижению скорости выполнения программы. Оптимизируйте приложение, чтобы снизить его энергопотребление и объем занимаемой им памяти, не забывая при этом о производительности.

3.6 Проектирование сервисных приложений

Сервис – это открытый интерфейс, обеспечивающий доступ к единице функциональности. Сервисы, буквально, обеспечивают некоторую программную услугу вызывающей стороне, потребляющей этот сервис.

Сервисы слабо связаны и для обеспечения более сложной функциональности могут комбинироваться с клиентом или в рамках других сервисов.

Сервисы могут быть распределенными, доступ к ним может осуществляться как с удаленного компьютера, так и с компьютера, на котором они выполняются.

Сервисы ориентированы на сообщения, т.е. интерфейсы сервисов описываются с помощью Web Services Description Language (WSDL), и операции вызываются с использованием схем сообщений на базе Extensible Markup Language (XML), передаваемых по транспортному каналу.

Сервисы поддерживают гетерогенную среду благодаря реализации возможности взаимодействия через описание сообщения/интерфейса. Если компоненты могут интерпретировать сообщение и описание интерфейса, они могут использовать сервис независимо от того, какая технология используется в их основе. На рис. 3.4 показано общее представление типовой архитектуры сервисного приложения.

Типовое сервисное приложение состоит из трех слоев: слой сервисов, бизнес-слой и слой доступа к данным.

Слой сервисов может включать компоненты интерфейсов сервисов, типов сообщений и типов данных; бизнес-слой - компоненты бизнес-логики, бизнес-процесса и бизнес-сущностей; и слой доступ к данным включает компоненты доступа к данным и агентов сервисов.

Сервисы являются гибкими по своей природе и могут использоваться в разнообразнейших сценариях и комбинациях.

Рассмотрим типовые сценарии:

– Сервис, предоставляемый через Интернет.

Этот сценарий описывает сервис, используемый рядом клиентов через Интернет. Сюда относятся как сервисы типа «бизнес-бизнес», так и ориентированные на потребителя сервисы. Биржевой Веб-сайт, использующий Веб-сервисы фондовой биржи и предоставляющий котировки акций - один из примеров такого сценария. Решения по аутентификации и авторизации должны приниматься, исходя из границ доверия в Интернете и типов учетных данных. Например, аутентификация по имени пользователя и паролю или применение сертификатов более вероятна в Интернет-сценарии, чем в интранет- сценарии.

– Сервис, предоставляемый по внутренней сети.

Этот сценарий описывает сервис, используемый по внутренней сети рядом (обычно ограниченным) внутренних или корпоративных клиентов. Система электронного документооборота уровня предприятия - один из примеров такого сценария. Решения по аутентификации и авторизации должны приниматься, исходя из границ доверия во внутренней сети и типов учетных данных.

Например, аутентификация Windows с применением Active Directory для хранения пользовательских данных является более вероятным для интранет-сценария, чем для Интернет-сценария.

– Сервис, предоставляемый локально на компьютере.

Этот сценарий описывает сервис, потребляемый приложением на локальном компьютере. Решения по защите на транспортном уровне и на уровне сообщений должны приниматься на основании локальных границ доверия и пользователей.

– Смешанный сценарий.

Этот сценарий описывает сервис, используемый множеством приложений по Интернету, внутренней сети и/или на локальном компьютере. Сервисное бизнес-приложение (LOB), потребляемое локально насыщенным клиентским приложением и Веб-приложением по Инернету - пример такого сценария.

Принципы проектирования

При проектировании сервисных приложений необходимо следовать общим рекомендациям, применимым ко всем сервисам. Такими рекомендациями являются проектирование слабо детализированных операций, следование контракту сервиса и готовность к возможному поступлению недействительных запросов или запросов в неверном порядке. Кроме общих рекомендаций, есть специальные рекомендации, которых следует придерживаться при проектировании разных типов сервисов. Например, для Сервисно-ориентированной архитектуры (SOA) необходимо обеспечить стабильность контракта приложения и автономность сервиса. Либо может создаваться приложение, предоставляющее сервисы рабочего процесса, или хранилище операционных данных, обеспечивающее основанный на сервисе интерфейс.

При проектировании сервисных приложений руководствуйтесь следующими рекомендациями:

– Используйте многослойный подход и избегайте тесного связывания слоев.

По возможности распределяйте бизнес-правила и функции доступа к данным по разным компонентам. Обеспечивайте интерфейс бизнес-слоя через абстракцию, которая может быть реализована путем использования открытых интерфейсов объектов, общих описаний интерфейсов, абстрактных базовых классов или через обмен сообщениями.

– Проектируйте слабо детализированные операции.

При работе с интерфейсом сервиса избегайте использования детализированных вызовов, что может привести к очень низкой производительности. Операции сервиса должны быть слабо детализированными и проецироваться на операции приложения. Применение шаблона Fagade позволит объединять несколько небольших детализированных операций в одну слабо детализированную операцию. Например, для статистических данных необходимо обеспечить операцию, которая будет возвращать все данные за один

вызов, чтобы не приходилось делать множество вызовов, каждый из которых будет возвращать лишь подмножество данных.

- При проектировании контрактов данных обеспечивайте возможность расширения и повторного использования.

Контракты данных должны быть спроектированы с обеспечением возможности их расширения без влияния на потребителей сервиса. Старайтесь создавать используемые сервисом составные типы из стандартных элементов. Слой сервисов должен знать о бизнес-сущностях, используемых бизнес- слоем. Обычно это обеспечивается путем создания для бизнес-сущностей отдельной сборки, которая совместно используется слоем сервисов и бизнес-слоем.

- При проектировании строго придерживайтесь контракта сервиса.

Слой сервисов должен реализовывать и обеспечивать только ту функциональность, которая оговорена контрактом сервиса, внутренняя реализация и детали сервиса никогда не должны предоставляться потребителям. Также если необходимо изменить контракт сервиса для включения в него новой функциональности, реализованной сервисом, и новые операции и типы не являются обратно совместимыми с существующими контрактами, создавайте новые версии контрактов. Описывайте новые операции, предоставляемые сервисом, в новой версии контракта сервиса и новые типы схем - в новой версии контракта данных.

- Проектируйте автономные сервисы.

Сервисы не должны предъявлять никаких требований к их потребителям и не должны делать никаких предположений о потребителе или о том, как он планирует использовать предоставляемый сервис.

- При проектировании должна быть учтена возможность поступления недействительных запросов.

Никогда не делайте предположения о том, что все поступающие на сервис сообщения будут действительными. Реализуйте логику валидации для проверки всех сообщений на соответствие схемам; и отвергайте или очищайте все недействительные сообщения. Обеспечьте сервису возможность выявлять и правильно обрабатывать повторные сообщения (идемпотентность) путем реализации широко известных шаблонов или через использование сервисов инфраструктуры. Кроме того, сервис должен уметь обрабатывать сообщения, поступающие в произвольном порядке (коммутативность). Это можно сделать, реализуя дизайн, позволяющий сохранять сообщения и затем обрабатывать их в правильном порядке.

- Сервисы должны управляться на основе политик и иметь явные границы.

Сервисное приложение должно быть самодостаточным и иметь четкие границы. Доступ к сервису должен быть разрешен только через слой интерфейса сервиса. Сервис должен публиковать политику, описывающую, как потребители могут взаимодействовать с сервисом. Это особенно важно для открытых сервисов, чтобы обеспечить потребителям возможность проверять политику и требования для взаимодействия.

- Разделяйте функциональность сервиса и операции инфраструктуры.

Логика сквозной функциональности никогда не должна смешиваться с логикой приложения, поскольку это может привести к реализациям, плохо расширяемым и сложным в обслуживании.

- Размещайте основные компоненты слоя сервисов в отдельных сборках.

Например, интерфейс, реализация, контракты данных, контракты сервисов, контракты сбоев и трансляторы - каждый из них должен размещаться в собственной сборке.

- Избегайте использования сервисов данных для предоставления отдельных таблиц базы данных.

Это приведет к слишком детализированным вызовам сервиса и зависимостям между операциями сервиса, что может создать проблемы зависимостей для потребителей сервиса. Кроме того, старайтесь избегать реализации бизнес-правил в сервисах, потому что разные потребители данных будут иметь собственные уникальные срезы доступных данных и правила, и это обусловит ограничения в использовании данных.

- Обеспечьте при проектировании, чтобы сервисы рабочего процесса использовали интерфейсы, поддерживаемые вашей подсистемой управления рабочим процессом.

Попытки создания собственных интерфейсов могут ограничить типы поддерживаемых операций, при этом потребуется больше усилий для расширения и обслуживания сервисов. Вместо того чтобы добавлять сервисы рабочего процесса в существующее сервисное приложение, спроектируйте автономный сервис, поддерживающий только требования рабочего процесса.

Заключение

Процесс проектирования архитектуры программного обеспечения состоит в проектировании структуры всех его компонент, функционально связанных с решаемой задачей, включая сопряжения между ними и требования к ним.

Архитектура программного обеспечения включает определение всех модулей программ, их иерархии и сопряжения между ними и данными.

Как иерархической системе архитектуре программного обеспечения присущ ряд свойств, важнейшими из которых являются:

- вертикальная соподчиненность, заключающаяся в последовательном упорядоченном расположении взаимодействующих компонент, составляющих данный комплекс программ;
- компоненты одного уровня обеспечивают реализацию функций компонент следующего уровня;
- каждый уровень иерархии реализуется через функции компонент более нижних уровней;
- каждый компонент знает о компонентах более низких уровней и ничего не знает о компонентах более высоких уровней;
- право вмешательства и приоритетного воздействия на компоненты любых уровней со стороны компонент более высоких иерархических уровней;

– взаимозависимость действий компонент верхних уровней от реакций на воздействия и от функционирования компонент нижних уровней, информация о которых передается верхним уровням.

Не следует смешивать понятия теории разработки ПО: архитектура и проектирование. Архитектура ПО является реализацией нефункциональных требований к системе, в то время как проектирование ПО является реализацией функциональных требований.

Архитектура ПО можно представить себе в виде разработки стратегии, т.е. деятельности, связанной с определением глобальных ограничений, накладываемых на проектирование системы, такие как выбор парадигмы программирования, архитектурных стилей, стандарты разработки ПО, основанные на использовании компонентов, принципы проектирования и ограничений, накладываемых государственным законодательством.

Детальное проектирование можно рассматривать как разработку тактики, т.е. деятельности, связанной с определением локальных ограничений проекта, такие как шаблоны проектирования, архитектурные модели, идиомы программирования и рефакторинга.

На практике архитектору приходится определять грань между архитектурой ПО (архитектурным дизайном) и детальным дизайном (неархитектурным проектированием), причем инструкций, как сделать это, которые подходили бы для любого случая нет. Архитектура является проектированием (дизайном), но не всякий дизайн является архитектурным дизайном

Литература

1. Фредерик П. Брукс. Проектирование процесса проектирования: записки компьютерного эксперта.: Пер. с англ. - М.: ООО "И.Д.Вильямс", 2012. – 464 с.
2. Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.: Пер. с англ. - М.: ООО "И.Д.Вильямс", 2011. – 448 с.
3. Фримен Э., Сьерра К., Б. Бейтс. Паттерны проектирования. – СПб.: Питер, 2011. – 656 с.
4. Спинеллис Д., Гусиос Г. Идеальная архитектура. Ведущие специалисты о красоте программных архитектур. Пер. с англ. – СПб.: Символ-Плюс, 2010. – 528 с.
5. Гудлиф П. Ремесло программиста. Практика написания хорошего кода. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 704 с.
6. Руководство Microsoft по проектированию архитектуры приложений. – 2-е издание., 2009. – 529 с.
7. Буч Г., РамбоД., Якобсон И. Введение в UML от создателей языка.2-е изд. – М.: ДМК Пресс, 2011. – 496 с.
8. Тидвелл ДЖ. Разработка пользовательских интерфейсов. 2-е изд. – СПб.: Питер, 2011. – 480 с.

9. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Библиотека программиста. – СПб.: Питер, 2010. – 368 с

10. Басс, Клементс, Кацман. Архитектура программного обеспечения на практике. - 2-е издание. СПб: Питер, 2006. – 575 с.

11. Фаулер М. Архитектура корпоративных программных приложений.: Пер. с англ. — М.: Вильямс, 2006. – 544 с.

12. Кролл П., Крачтен Ф. Rational Unified Process – это легко. Руководство по RUP. Пер. с англ. – М.:КУДИЦ-ОБРАЗ, 2004. – 423 с.

13. Галерея успешных архитектур. [Электронный ресурс]. Режим доступа: http://www.splc.net/fame.html .

14. Кулямин В. В. Технологии программирования. Компонентный подход. [Электронный ресурс]. Режим доступа: http://lib.mdpu.org.ua/e-book/vstup/L/Jogolev.pdf.

15. Амблер С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. – СПб.: Питер, 2005. – 412 с.

16. Вигерс К. Разработка требований к программному обеспечению/Пер. с англ. – М.: Русская Редакция, 2004. – 576 с.

17. ANSI/IEEE 1471-2000

18. Рекомендованная практика описания архитектуры преимущественно программных систем. [Электронный ресурс]. Режим доступа: http://standards.ieee.org/findstds/standard/1471-2000.html.

19. ГОСТ Р ИСО/МЭК 15288—2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем.

20. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. - Пер. с англ. - СПб.: Символ-Плюс, 2011. - 768 с.