GeekBrains

Архитектура ПО
Урок 5
Горизонтальные уровни и вертикальные срезы архитектуры

Оглавление

Введение

Архитектор стремится получить структуру системы, поддерживающую требуемые варианты использования, но не может знать их все и сразу. Зато он всегда знает основное назначение системы, так как без этого знания начинать разработку системы бессмысленно.

Это может быть онлайн-магазин, система учёта материалов или система обработки заказов. Поэтому, опираясь на контекст назначения системы, архитектор использует принципы единственной ответственности и согласованного изменения для отделения друг от друга всего, что изменяется по разным причинам.

Такой подход позволит структурировать понимание системы на высоком уровне, понять распределение обязанностей в проектируемой системе, выбрать подходящие технологии для каждого уровня, а также учесть сценарии использования системы.

На этом уроке

1. Разберём понятие горизонтальных и вертикальных слоёв приложения.
2. Вспомним «контрактное программирование» и рассмотрим подход «защитного программирования».
3. Узнаем, что такое распределённые системы.
4. Изучим паттерны доступа к данным.

1. Понятие горизонтальных уровней архитектуры и смысл их разделения

Практически любую систему, взаимодействующую с пользователем, можно разделить на логические слои. Также можно выделить сценарии-срезы связывания такой системы для выполнения конкретных задач.

Слои и срезы — это следующий уровень абстракции после компонентов.

1. **Горизонтальные слои** связаны с назначением слоя в коде системы. Например, есть слой базы данных и бизнес-логики.
2. **Вертикальные слои** связаны с тем, что делает система, с её функциями (сценарии использования). Сценарии использования меняются по разным причинам, друг от друга независимым. Поэтому важно защищать их друг от друга.

Слои обеспечивают логический уровень разделения в приложении. Если логика приложения при этом физически распределяется между несколькими серверами или процессами, такие раздельные физические целевые объекты развёртывания называются **уровнями**.

Такие принципы деления системы применяются также для объединения всего, что изменяется по одной причине.

Систему всегда можно разделить на горизонтальные уровни:

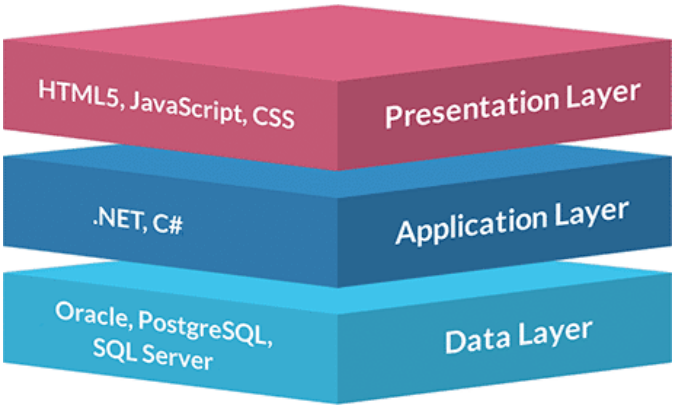
- пользовательский интерфейс;
- бизнес-правила, характерные для приложения;
- бизнес-правила, не зависящие от приложения;
- слой базы данных.

Пользовательский интерфейс — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее — область человеческой деятельности, поддерживаемая системой.

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Программные системы, разбитые на ряд слоёв, рисуются в виде горизонтальных прямоугольников, например, как на диаграмме ниже, и представляют различные высокоуровневые логические компоненты системы.



Пример многослойной архитектуры

Преимущества использования горизонтальных уровней

1. Горизонтальные уровни разделяются по вышеупомянутым слоям приложения. Они разделяют рабочую нагрузку и ресурсы разработки между отдельными фрагментами. Этот подход, безусловно, выглядит логичным: каждому фрагменту назначены специализированные ресурсы для разработки.

2. Многие разработчики имеют специализацию в разработке ПО и не знают всего технологического стека. Строго определённый технологический стек даёт программистам больше возможностей для производства лучшего продукта.

3. Горизонтальные слои позволяют развивать, развёртывать и поддерживать себя независимо друг от друга. Например, пользовательский интерфейс отделён от бизнес-логики. Изменение версии пользовательского интерфейса — это просто замена файла jar. Хорошее горизонтальное разделение — то, при котором бизнес-правила не знают о пользовательском интерфейсе.

4. Организуя код в слои, общая функциональность может повторно использоваться во всём приложении. Это повторное использование полезно, оно означает, что требуется писать меньше кода, следуя принципу «не повторять себя» (DRY).

5. Через многоуровневые архитектуры приложения налагаются ограничения на то, какие слои могут взаимодействовать с другими слоями. Этот подход позволяет достичь лучшей инкапсуляции.

6. Слои и инкапсуляция значительно облегчают замену функциональности в приложении. Например, приложение изначально использует собственную базу данных SQL Server для сохранения, а позже применяет облачную стратегию персистентности или стратегию веб-API.

Негативные стороны горизонтальных уровней

1. Вероятно, одна из самых больших проблем заключается в том, что интеграция фрагментов часто не может начаться до тех пор, пока большинство или все компоненты не завершатся.

2. Причина, по которой интеграция к концу разработки компонентов превращается в проблему, заключается в том, что на тестирование выделяется меньше времени. Такой короткий срок усугубляется дополнительной рабочей нагрузкой, введённой обратной связью клиентов или заинтересованных сторон. И нередко устанавливаются нереальные сроки для этого этапа проекта.

При разработке многоуровневого программного приложения, особенно в начале проекта, обычно применяют два подхода.

1 Подход) Можно строить горизонтальные слои один за другим, как правило, начиная с самого низкого уровня. Например, данные системы сначала представляют в виде схемы базы данных. Далее для доступа к этим данным записывают набор функций доступа к данным.

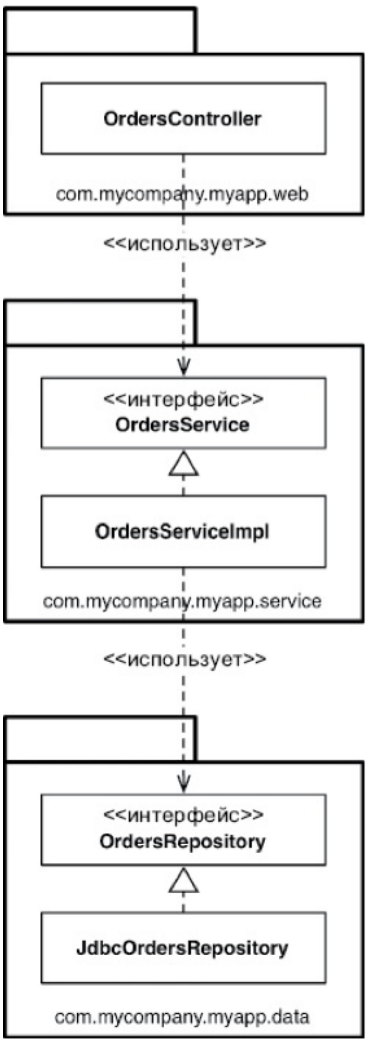
Затем добавляют бизнес- или сервисный уровень, куда включают дополнительную функциональность. Наконец, можно создать пользовательский интерфейс. На каждом шаге разработчики выбирают создание многоразовых библиотек, компонентов или виджетов.

2 Подход) Горизонтальный подход не всегда идёт снизу вверх — можно начать с пользовательского интерфейса и далее работать вниз.

1.1. Уровень пользовательского интерфейса

Пользовательский интерфейс изменяется по причинам, как правило, не имеющим ничего общего с бизнес-правилами (но не всегда). Архитектор отделяет часть, отвечающую за пользовательский интерфейс, от части, реализующей бизнес-правила. Поэтому их можно изменять независимо друг от друга, сохраняя ясную видимость вариантов использования.

На рисунке ниже демонстрируется OrdersController — веб-контроллер, обрабатывающий запросы из сети.



Пример разделения на горизонтальные уровни

Пользовательский интерфейс объединяет в себе все элементы и компоненты программы, способные влиять на взаимодействие пользователя с программным обеспечением (ПО).

К этим элементам относятся:

- Задачи пользователя, решаемые через систему.
- Элементы управления системой.
- Навигация между блоками системы.
- Дизайн экранов программы.
- Средства отображения информации, отображаемая информация и форматы.
- Устройства и технологии ввода данных.
- Обратная связь с пользователем.
- Порядок использования программы и документация к ней.

Что содержится:

- Веб-страницы (MPA).
- Веб-приложение (SPA).
- Мобильное приложение для Android или iOS.
- Десктопное приложение для Windows.
- Технический пользовательский интерфейс.

Что такое UX/UI?

User Experience — пользовательский опыт, то, как интерфейс ведёт себя с пользователем.

Простыми словами, это то, каким образом пользователь взаимодействует с интерфейсом и насколько сайт или приложение понятны и удобны пользователю.

Понятие UX впервые было сформулировано только в начале 90х. Его придумал психолог и проектировщик Дональд Норман, который присоединился к команде Apple.

Он описал этот термин в своей книге «Дизайн привычных вещей». Компания Apple с самого начала уделяла большое внимание пользовательскому взаимодействию, и её интерфейсы считаются одними из лучших.

UX включает в себя навигацию по сайту, функциональность меню и результат взаимодействия со страницами. Это не только «костяк» сайта — его структура, — но и коммуникация: диалоговые окна, функциональность кнопок, настройки поиска и форм.

User Interface — пользовательский интерфейс, то, как интерфейс выглядит для пользователя — его **дизайн, User design.**

Проще говоря — оформление сайта: сочетания цветов, шрифты, иконки и кнопки.

Одна часть не может существовать без другой.

UX — это функции интерфейса,

UI — его внешний вид.

Проектирование визуальной части дизайна, UX/UI, — это один из этапов создания как интерфейса, так и системы.

1.2. Уровни бизнес-логики

Бизнес-уровень содержит основные функциональные возможности приложения, включая большую часть бизнес-логики обработки и манипулирования данными, полученную с уровня данных. Как часть «бэкенда» приложения, этот слой представляется средним уровнем между презентационным слоем и уровнем данных.

На рисунке представлен OrdersService — интерфейс, определяющий бизнес-логику, связанную с заказами. OrdersServiceImpl — реализация службы заказов.
Есть более или менее тесно связанные с приложением бизнес-правила. Например, проверка полей ввода — это бизнес-правило, тесно связанное с приложением.

Вычисление процентов по вкладу и подсчёт запасов, напротив, — бизнес-правила, более тесно связанные с предметной областью.

Эти виды бизнес-правил меняются с разной скоростью и по разным причинам, поэтому и разделить правила надо так, чтобы изменять независимо. То есть уровень бизнес-логики делится на 2 подуровня: бизнес-логику приложения и бизнес-логику предметной области.

Уровень OrdersService определяет основную бизнес-логику, куда входят:

- аутентификация и авторизация пользователей;
- основная логика бизнеса или приложений;
- интеграция со сторонними сервисами;
- вызовы прямо на уровень данных.

1.3. Domain Driven Design (DDD)

Domain-driven design, или предметно-ориентированное проектирование, — это концепция, согласно которой структура и язык программного кода (имена классов, методы классов, переменные классов) должны соответствовать бизнес-области. Данный термин был впервые введён Э. Эвансом в книге, которая так и называется: *Domain-Driven Design*.

Единый язык предметной области

При разработке программного обеспечения постоянно возникает проблема с пониманием кода: что он собой представляет, что он делает, как и почему. Ещё сложнее понять код, если в нём используется терминология, отличная от терминологии экспертов домена. Например, если эксперты домена говорят о старших пользователях, а код говорит о супервайзерах, это может внести путаницу в обсуждение приложения. Однако большую часть этой двусмысленности можно устранить с помощью правильного именования классов и методов: они должны выражать, чем является объект и что делает метод в контексте домена. Основная идея использования единого языка — привести приложение в соответствие с бизнесом. Это достигается путём принятия общего языка между бизнесом и технологией в коде.

Источником *единого языка* является бизнес-сторона компании (например торговля-ритейл, медицина, неефтегазовая пормышленность), у них есть концепции, которые необходимо реализовать, но терминология затем согласовывается с технологической стороной компании (это означает, что бизнес-сторона не всегда выбирает лучшее название) с целью создания общей терминологии, которая может использоваться бизнесом, технологией и в самом коде без каких-либо двусмысленностей.

Код, классы, методы, свойства и наименование модулей должны соответствовать единому языку. При необходимости стоит провести рефакторинг кода!

Очень важно понимать, что в рамках предметной области смысл определённого термина или фразы может сильно отличаться. Существует некая граница, в пределах которой понятия единого языка имеют вполне конкретное контекстное значение.

Ограниченный контекст

Эта концептуальная граница называется ограниченный контекст (Bounded context). Это второе по значимости свойство DDD после единого языка. Оба эти понятия взаимосвязаны и не могут существовать друг без друга.

- В каждом ограниченном контексте существует только один единый язык.
- Ограниченные контексты являются относительно небольшими, меньше, чем может показаться на первый взгляд. Ограниченный контекст достаточно велик только для единого языка изолированной предметной области, но не больше.
- Единый — значит «вездесущий» или «повсеместный», т. е. язык, на котором говорят члены команды и на котором выражается отдельная модель предметной области, разрабатываемой командой.
- Язык является единым только в рамках команды, работающей над проектом в едином ограниченном контексте.
- Попытка применить *единый язык* в рамках всего предприятия или, что хуже, среди нескольких предприятий, закончится провалом.

Предметная область, предметная подобласть, смысловое ядро

Предметная область (Domain) — это то, что делает организация, и среда, в которой она это делает. Разработчик программного обеспечения для организации обязательно работает в её предметной области. Следует понимать, что при разработке модели предметной области необходимо сосредоточиться в определённой подобласти, так как практически невозможно создать единственную, всеобъемлющую модель бизнеса даже умеренно сложной организации. Очень важно разделять модели на логические разделённые *предметные подобласти* (Subdomain) всей предметной области, согласно их фактической функциональности. Подобласти позволяют быстрее определить разные части предметной области, необходимые для решения конкретной задачи.

Также необходимо уметь определять *смысловое ядро* (Core domain). Это очень важный аспект подхода DDD. Смысловое ядро — это подобласть, имеющая первостепенное значение для организации (банковсие операции для банка, торговля для маркетплейса). Со стратегической точки зрения бизнес должен выделяться своим смысловым ядром. Большинство DDD-проектов сосредоточены именно на смысловом ядре. Лучшие разработчики и эксперты должны быть задействованы именно в этой подобласти. Большая часть инвестиций должна быть направлена именно сюда для достижения преимуществ для бизнеса и получения наибольшей прибыли.

Если моделируется определённый аспект бизнеса, который важен, но не является смысловым ядром, то он относится к *служебной подобласти* (Supporting subdomain). Бизнес создаёт служебную подобласть, потому что она имеет специализацию. Если она не имеет специального предназначения для бизнеса, а требуется для всего бизнеса в целом, то её называют *неспециализированной подобластью* (Generic subdomain). Эти виды подобластей важны для успеха бизнеса, но не имеют первоочередного значения. Именно смысловое ядро должно быть реализовано идеально, поскольку оно обеспечивает преимущество для бизнеса.

1.4. Уровень хранения данных

Уровень данных контролирует хранение и распределение данных, как правило, в таблицах баз данных. Хотя базы данных используют и не реляционные методы представления информации.

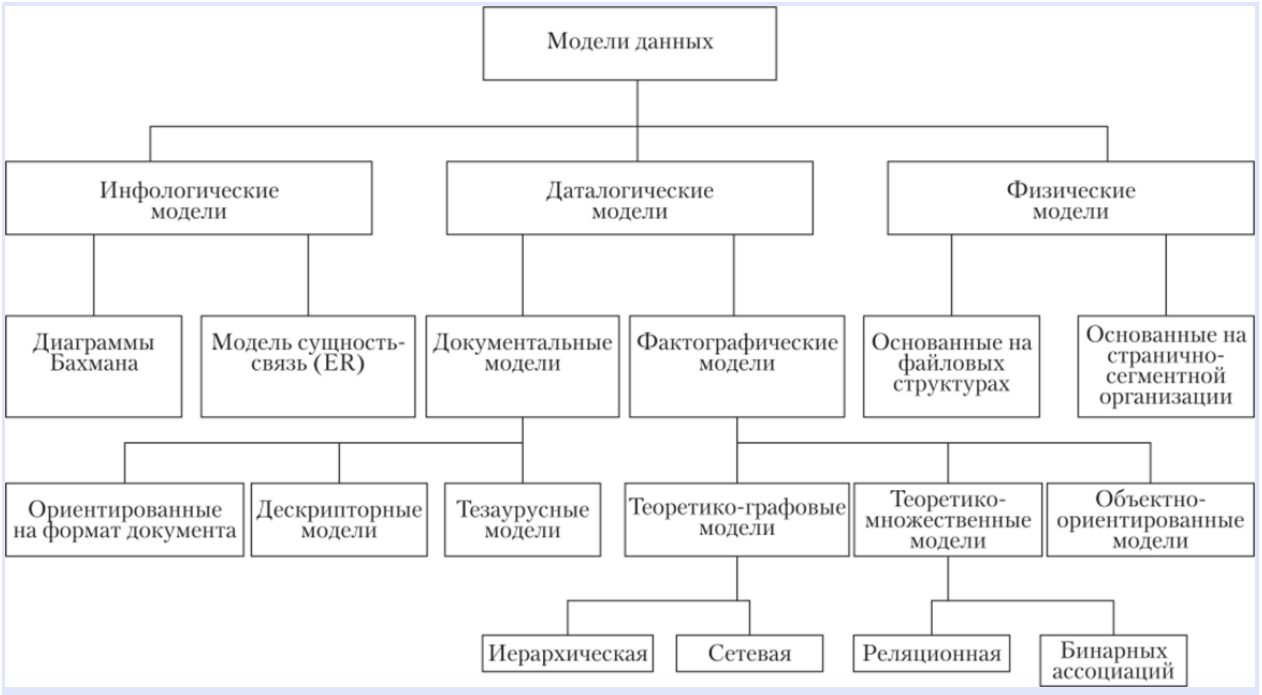
База данных, язык запросов и даже схема — это технические детали, которые имеют мало общего с бизнес-правилами и пользовательским интерфейсом. Они изменяются со скоростью и по причинам, не зависящим от других аспектов системы. Следовательно, в архитектуре они отделяются от остальной системы, чтобы изменяться их независимо.

Из примера на рисунке 2 OrdersRepository — это интерфейс, определяющий порядок доступа к информации о заказах в хранилище. JdbcOrdersRepository — реализация интерфейса хранилища.

Сюда входят:

- модели данных;
- запросы;
- доступ к базе данных;
- миграции.

Модель данных — это некоторая абстракция, которая, будучи применима к конкретным данным, позволяет пользователям и разработчикам трактовать их уже как информацию, то есть сведения, содержащие не только данные, но и взаимосвязь между ними.



Классификация моделей данных

Модели, выражающие информацию о предметной области в виде, независимом от используемой системой управления базами данных (СУБД), называются **инфологическими или семантическими**. Они отражают в естественной и удобной для разработчиков и других пользователей форме информационно-логический уровень абстрагирования, связанный с фиксацией и описанием объектов предметной области, их свойств и их взаимосвязей.

Инфологические модели данных используются на ранних стадиях проектирования, чтобы описывать структуры данных в процессе разработки приложения. **Даталогические** модели уже поддерживаются конкретной СУБД.

В настоящий момент в качестве **физических** моделей используются различные методы размещения данных, основанные на файловых структурах. Это организация:

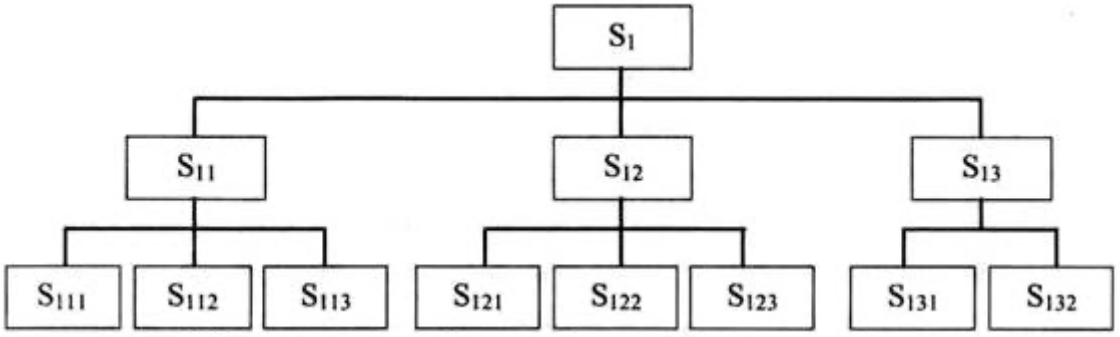
- файлов прямого и последовательного доступа;
- индексных и инвертированных файлов;
- файлов, использующих различные методы кэширования;
- взаимосвязанных файлов.

Современные СУБД широко используют страничную организацию данных. Физические модели данных, основанные на страничной организации, считаются наиболее перспективными.

В зависимости от вида организации данных в группе фактографических моделей данных различаются следующие основные модели СУБД:

- иерархическая;
- сетевая;
- реляционная;
- объектно-ориентированная.

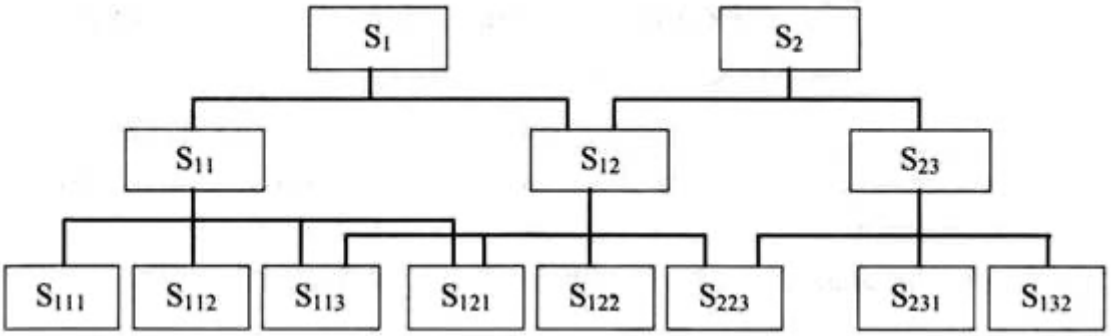
Иерархические модели СУБД имеют древовидную структуру. Каждому узлу структуры соответствует один сегмент, представляющий собой перечисленный линейный кортеж полей данных. А каждому сегменту соответствует один входной и несколько выходных сегментов.



Иерархическая модель СУБД

Сетевая даталогическая модель СУБД во многом подобна иерархической. Отличие заключается в том, что в сетевой модели для сегментов допускается несколько входных сегментов наряду с возможностью применения сегментов без входов, с точки зрения иерархической структуры.

Таким образом, под сетевой СУБД понимается система, поддерживающая сетевую организацию: любая запись, называемая записью старшего уровня, может содержать данные, которые относятся к набору других записей, называемых записями подчинённого уровня.



Сетевая модель СУБД

Реляционная модель — это базовая и наиболее привычная форма представления данных «в виде таблиц». В теории множеств таблице соответствует термин «отношение» (relation), который и дал название реляционной алгебре, описывающей реляционную модель.

ПРЕП.

НП
П1
П2
П3

ЧИТАЕТ

НП	НК
П1	К1
П1	К2
П1	К3
П2	К1
П2	К3
П3	К2
П3	К3

КУРС

НК
К1
К2
К3

Реляционная модель СУБД

Такая модель позволяет определять:

1. Операции по запоминанию и поиску данных:

   a. операции на множествах: объединение, пересечение, разность, произведение;
   b. реляционные операции: выбрать, спроецировать, соединить, разделить.
2. Ограничения, связанные с обеспечением целостности данных.

Объектно-ориентированная модель расширяет определение сущности, чтобы включить в него не только атрибуты, описывающие состояние объекта, но и действия, которые с ним связанные, т. е. его поведение. В таком случае говорится, что объект инкапсулирует состояние и поведение.

2. Варианты использования — вертикальные срезы

2.1. Вертикальные срезы

Если требуется добавить новую форму в веб-интерфейс, то надо сделать новую view, потом controller, добавить model, поменять что-то в слое доступа к данным, добавить новую таблицу в базу данных и подключить к ней классы доступа.

Новый инструментарий возникает нечасто. Поэтому в основном меняем то, что уже реализовалось. Но чтобы «немного» поправить текущий инструментарий и добавить новое «небольшое» поле, надо поправить view, потом controller, изменить model, поменять что-то в слое доступа к данным, обновить таблицу в базе данных и классы доступа к ней.

Если исключить небольшой «инфраструктурный» период в начале проекта, то основные задачи команды по добавлению нового или развитию имеющегося инструментария станут сквозными и затронут множество слоёв приложения.

В архитектуре горизонтальные слои, как правило, выражены чётко, а интерфейсы, представляющие конкретные слои, могут быть весьма массивными, со скрытием компонентов нижних уровней.

Решение

Горизонтальные слои, как правило, вертикально делятся на группы компонентов в соответствии с различными сценариями использования, которые обычно имеет приложение. Это может выглядеть как многослойный торт, где каждый ломтик — пример (сценарий) использования, а каждый слой в ломтике — компонент. Варианты использования дают естественный способ деления системы. При развёртывании приложения компоненты группируются любым способом, который имеет наибольший смысл.

Например, на видеосайте одним из примеров использования считается плеер видео. Таким образом, в этом приложении есть компоненты: ViewerUseCase, ViewerPresenter, ViewerView и прочие.

Другой сценарий использования — для издателя, который загружает видео на сайт. Для них предусматриваются компоненты: PublisherUseCase, PublisherPresenter, PublisherView и так далее. Дополнительный сценарий использования может быть и для администратора сайта.

Таким образом, варианты использования образуют узкие вертикальные срезы, пересекающие горизонтальные слои системы. Каждый вариант использования включает:

- какую-то долю пользовательского интерфейса;
- часть бизнес-правил, характерных для приложения;
- какие-то бизнес-правила, не зависящие от приложения;
- некоторые функции базы данных.

То есть вместе с делением системы на горизонтальные слои, мы одновременно делим её на тонкие вертикальные варианты использования, пересекающие эти слои.

Разделив элементы системы, изменяющиеся по разным причинам, мы можем добавлять новые варианты использования, не влияя на имеющиеся.

Если сгруппировать поддержку этих случаев использования в пользовательском интерфейсе и в базе данных так, что для всех них в пользовательском интерфейсе и базе данных будут предусматриваться свои аспекты, то добавление новых случаев использования почти гарантированно не повлияет на имеющиеся.

Концепция разбивки проектной работы на вертикальные фрагменты часто ассоциируется с процессами Scrum и Agile и представляет собой более современный подход к разбивке задач на более мелкие, более управляемые части. Поскольку фрагменты делят слои приложения вертикально и включают в себя всю функциональность конкретной функции от пользовательского интерфейса до уровня хранения данных. Вертикальные срезы должны быть небольшими. После завершения работы она сразу же будет готова к интеграции и тестированию.

Этот подход имеет преимущества в разработке программного обеспечения.

1. Быстрая обратная связь.
2. Быстрое обнаружение и разрешение дефектов.
3. Адаптируемость к изменениям в требованиях.
4. Небольшие части также легче определить, оценить и протестировать.

Вертикальная нарезка также сопряжена с множеством собственных проблем из-за дополнительной сложности процесса и архитектуры.

1. С командами, разделёнными на несколько наборов технологий — фронтенд, бэкенд и всё, что между ними — разработчики должны быть более кросс-функциональными. Вертикальный нарез популярен среди технологических стартап-компаний.
2. Ситуация, к которой трудно подойти «с вертикальным разделением», заключается в том, что, как правило, есть большие устаревшие приложения (легаси). Эти приложения распределяются по всему предприятию. Перемещение больших команд разработчиков на новый процесс в сложных условиях — уже нетривиальная задача.

2.2. Режимы разделения вариантов использования

Разберём, какое значение имеет режим разделения для архитектур систем.

Если разные варианты использования разделились, тогда те, что работают с максимальной пропускной способностью, уже наверняка отделились от работающих с низкой пропускной способностью. Так как это разные сценарии использования.

А если пользовательский интерфейс и база данных отделились от бизнес-правил, значит, компоненты, их реализующие, могут выполняться на разных узлах вычислительной системы — серверах. Части, которые выполняются с максимальной пропускной способностью, можно запускать на нескольких серверах.

Проще говоря, разделение вариантов использования также позволяет повысить эффективность работы системы. Однако для этого разделение должно иметь соответствующий режим.

Чтобы разделённые компоненты выполнялись на собственных серверах, они не должны зависеть от наличия общего адресного пространства в памяти и других ресурсов операционной системы. Их задача — быть независимыми службами, взаимодействующими исключительно по сетевым протоколам.

Многие архитекторы называют такие компоненты «службами» («сервисами») или «микрослужбами» («микросервисами»). Эти компоненты зависят от некоторого слабо формализуемого понятия «величины». Архитектуру, основанную на службах (или сервисах), часто называют сервис-ориентированной архитектурой.

Хорошая архитектура как можно дольше оставляет возможности открытыми. Режим разделения — одна из таких возможностей. И в случае грамотной декомпозиции системы на компоненты это не такая уж и сложная задача.

Есть разные способы деления на уровни и варианты использования, декомпозиции на компоненты. Деление выполняется на трёх уровнях:

- исходного кода;
- двоичного кода (развёртывания);
- единиц выполнения (служб).

Уровень исходного кода

Можно так настроить зависимости между модулями с исходным кодом, чтобы изменения в одном модуле не вынуждали производить изменения в других.

При использовании этого режима разделения все компоненты выполняются в общем адресном пространстве и взаимодействуют, просто вызывая функции друг друга. То есть имеется единственный исполняемый файл, загружаемый в память компьютера. Часто такой подход называется монолитной структурой.

Уровень развёртывания

Можно так настроить зависимости между единицами развёртывания, jar-файлами или динамически загружаемыми библиотеками dll, чтобы изменения

в исходном коде в одном модуле не вынуждали производить повторную сборку и развёртывание других.

Многие компоненты иногда находятся в общем адресном пространстве и взаимодействуют, вызывая функции друг друга. Другие компоненты выполняются в других процессах на той же машине и взаимодействуют посредством механизмов межпроцессных взаимодействий, сокетов или разделяемой памяти. Важно отметить, что в этом случае разделённые компоненты находятся в независимых единицах развёртывания, таких как jar-, gem-файлы или динамически загружаемые библиотеки dll. Такую программную структуру ещё называют монолитной.

Уровень служб

Можно ограничить зависимости до уровня структур данных и взаимодействовать, обмениваясь исключительно сетевыми пакетами, чтобы каждая единица выполнения не зависела от изменений в исходном и двоичном коде в других, например, службы и микрослужбы. Важно, чтобы эти единицы выполнялись независимо на разных вычислительных машинах.

Стоит отметить, что рассмотренные режимы разделения практически не отличаются в архитектурном смысле — в плане декомпозиции на компоненты, выделения слоёв и срезов. При грамотном разделении возможен плавный переход между этими режимами.

3. Понятие распределённой системы

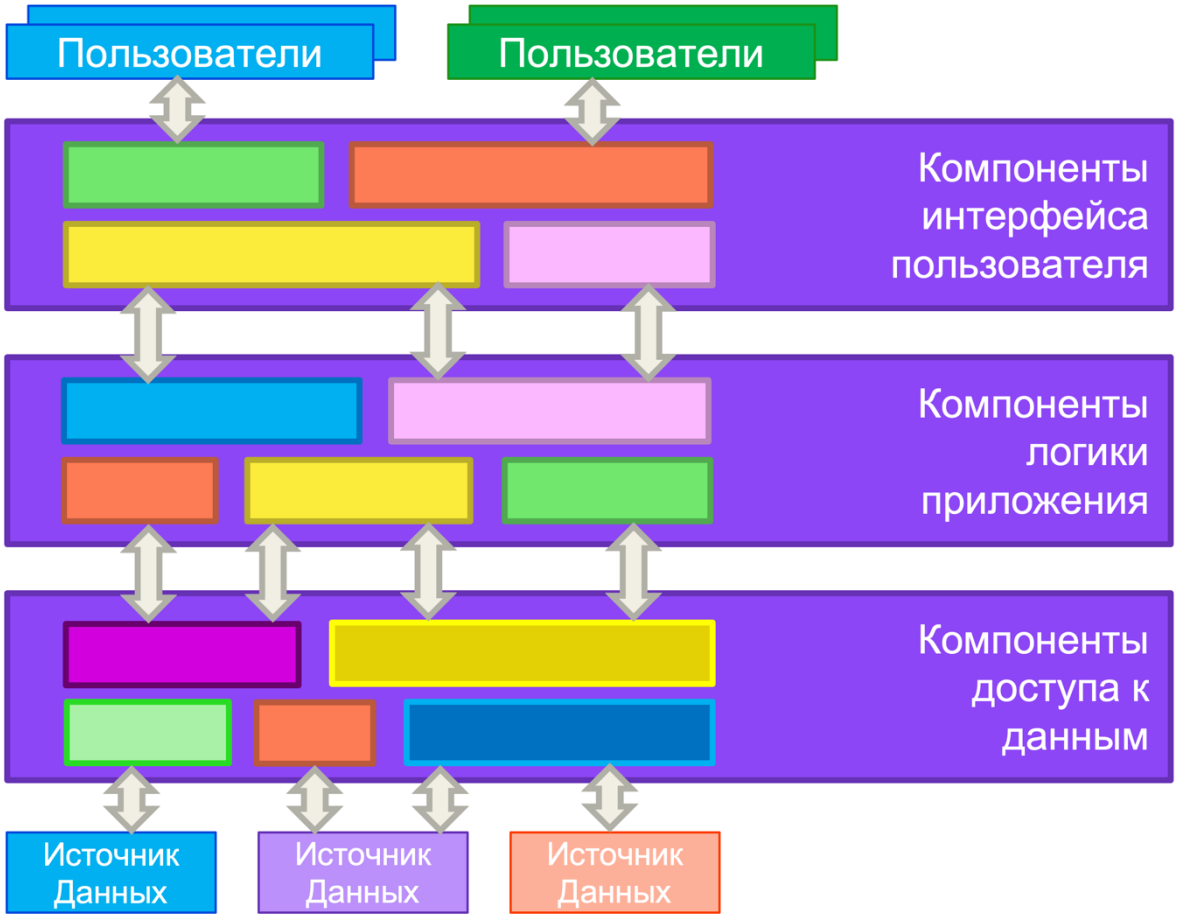
3.1. Архитектуры независимых компонентов

Программный компонент — программная часть системы компонент программного обеспечения.

Применительно к информационным системам распределёнными обычно называют системы с логикой приложения, распределённой между несколькими компонентами системы, каждый из которых выполняется на отдельном компьютере. Однако определение распределённой системы как совокупности компьютеров считается условным, поскольку программное обеспечение, установленное на одном компьютере, может отвечать за выполнение функций, относящихся к разным уровням. Для описания и реализации распределённых систем вводится понятие программного компонента.

Программная компонента — это единица программного обеспечения, исполняемая на одном компьютере в пределах одного процесса и предоставляющая некоторый набор сервисов. Эти сервисы используются через внешний интерфейс программной компоненты другими компонентами как выполняющимися на этом же компьютере, так и на удалённых компьютерах. Ряд компонент пользовательского интерфейса предоставляют свой сервис конечному пользователю.

Таким образом, распределённая система — это набор взаимодействующих программных компонентов, выполняющихся на одном или нескольких связанных компьютерах и выглядящих, с точки зрения пользователя системы, как единое целое.



Компоненты информационной системы

Компоненты — это единицы развёртывания. Во время модернизации системы одни компоненты обновляются независимо от прочих компонент. В хорошо спроектированной системе функции каждой компоненты относятся только к одному уровню приложения. Компоненты состоят из хорошо спроектированных программных структур среднего уровня (классов).

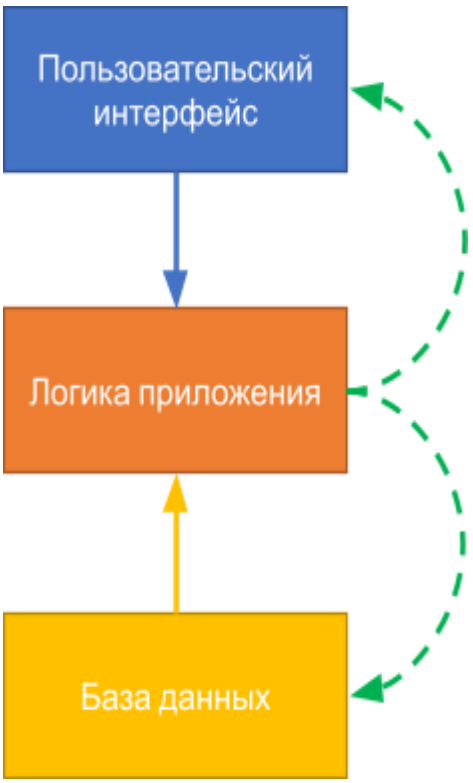
Компоненты представляют наименьшие сущности, которые развёртываются в составе системы. В Java — это jar-файлы. В Ruby — gem-файлы. В .Net — библиотеки DLL. В компилирующих языках — комплексы двоичных файлов. В интерпретирующих языках — комплексы файлов с исходным кодом.

Компоненты обычно объединяются в один выполняемый файл, собираются в один архив, например, файл .war, или развёртывается независимо, как отдельные плагины, загружаемые динамически: файлы .jar, .dll или .exe. Но независимо от способа развёртывания, правильно спроектированные компоненты всегда сохраняют возможность **независимого развёртывания** и, соответственно, **разрабатываются независимо**.

Рассмотрим **пример**.

Пусть в системе есть 3 компонента: пользовательский интерфейс (ПИ), бизнес-правила и база данных. Представим диаграмму так, что база данных и пользовательский интерфейс в системе будут зависеть от бизнес-правил.

То есть в исходном коде с реализацией бизнес-правил могут отсутствовать любые ссылки на ПИ или базу данных.



Пример системы, состоящей из трёх компонентов

Как следствие, бизнес-правила, ПИ и база данных компилируются в три разных компонента или единицы развёртывания, например, jar-файлы, библиотеки DLL или файлы Gem.

Как результат, появляется возможность развёртывать бизнес-правила независимо от ПИ и базы данных. Изменения в ПИ или в базе данных не будут оказывать никакого влияния на бизнес-правила. Проще говоря, когда реализация компонента изменится, достаточно повторно развернуть только этот компонент. Это называется **независимостью развёртывания.**

Если система состоит из модулей, которые развёртываются независимо, они разрабатываются независимо, разными командами. Это называется **независимостью разработки.**

Компонентная разработка (CBD) — это процедура, которая подчёркивает проектирование и разработку компьютерных систем с помощью повторно используемых программных компонентов.

В CBD акцент смещается с программирования на составление программных систем.

3.2. Распределённые системы

Программные системы, архитектура которых состоит из независимо работающих компонентов, общающихся друг с другом — называются распределёнными.

Распределённая система, это система, для которой отношения местоположений элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а следовательно, и с точки зрения анализа и синтеза системы.

Для распределённых систем характерно распределение функций, ресурсов между множеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы.

Основные преимущества распределённых приложений

Хорошая масштабируемость. При необходимости вычислительная мощность распределённого приложения может быть легко увеличена без изменения его структуры, система может легко интегрировать в своей транспортной среде новые вычислительные ресурсы.

Возможность управления нагрузкой. Промежуточные уровни распределённого приложения дают возможность управлять потоками запросов пользователей и перенаправлять их менее загруженным серверам для обработки;

Глобальность. Распределённая структура позволяет следовать пространственному распределению бизнес-процессов и создавать клиентские рабочие места в наиболее удобных точках.

Распределённые вычислительные системы обладают такими общими свойствами, как:

- *Управляемость.* Подразумевает способность системы эффективно контролировать свои составные части. Это достигается благодаря использованию управляющего ПО.
- *Производительность.* Обеспечивается за счёт возможности перераспределения нагрузки на серверы системы с помощью управляющего ПО;
- *Расширяемость.* К распределённым приложениям можно добавлять новые составные части (серверное ПО) с новыми функциями.

Почти всегда при разработке новой информационной системы можно найти подходящие по своим функциям уже работающие готовые компоненты. Проблема состоит в том, что при их создании не учитывались требования интероперабельности. Эти компоненты не понимают один другого, они не могут работать совместно.

Интероперабельность, или функциональная совместимость (от англ. *interoperability* — «способность ко взаимодействию») — это способность продукта или системы, интерфейсы которых полностью открыты, взаимодействовать и функционировать с другими продуктами или системами без каких-либо ограничений доступа и реализации.

Желательно иметь механизм или набор механизмов, которые позволят сделать такие независимо разработанные информационно-вычислительные ресурсы интероперабельными.

Компонент, называемый клиентом, выдаёт запрос для выполнения какого-либо действия к другому компоненту, называемому сервером. Такое отношение называется клиент-серверным. Интерфейс сервера должен быть собран в одном месте и чётко определён. Преобладание таких отношений характерно для клиент-серверной архитектуры.

Классическая двухуровневая архитектура клиент-сервер постоянно усложнялась добавлением новых уровней. Отдельный уровень может управлять процедурами (сервер приложений), другой — данными (сервер баз данных в трехуровневой архитектуре).

Одно из наиболее признанных решений проблемы унаследованных систем основывается на объектно-ориентированном подходе.

3.3. Требования к распределённым системам

Чтобы достигнуть цели своего существования — улучшения выполнения запросов пользователя — распределённая система должна удовлетворять некоторым необходимым требованиям. Можно сформулировать следующий набор требований, которым в наилучшем случае должна удовлетворять распределённая вычислительная система.

Открытость

Все протоколы взаимодействия компонент внутри распределённой системы в идеальном случае должны быть основаны на общедоступных стандартах. Это позволяет использовать для создания компонент различные средства разработки и операционные системы. Каждая компонента должна иметь точную и полную спецификацию своих сервисов. В этом случае компоненты распределённой системы могут быть созданы независимыми разработчиками. При нарушении этого требования может исчезнуть возможность создания распределённой системы, охватывающей несколько независимых организаций.

Масштабируемость

Масштабируемость вычислительных систем имеет несколько аспектов. Наиболее важный из них для — возможность добавления в распределённую систему новых компьютеров для увеличения производительности системы, что связано с понятием балансировки нагрузки (load balancing) на серверы системы. К масштабированию относятся также вопросы эффективного распределения ресурсов сервера, обслуживающего запросы клиентов.

Поддержание логической целостности данных

Запрос пользователя в распределённой системе должен либо корректно выполняться целиком, либо не выполняться вообще. Ситуация, когда часть компонент системы корректно обработали поступивший запрос, а часть — нет, является наихудшей.

Устойчивость

Под устойчивостью понимается возможность дублирования несколькими компьютерами одних и тех же функций или же возможность автоматического распределения функций внутри системы в случае выхода из строя одного из компьютеров. В идеальном случае это означает полное отсутствие уникальной точки сбоя, то есть выход из строя одного любого компьютера не приводит к невозможности обслужить запрос пользователя.

Безопасность

Каждый компонент, образующий распределённую систему, должен быть уверен, что его функции используются авторизированными на это компонентами или пользователями. Данные, передаваемые между компонентами, должны быть защищены как от искажения, так и от просмотра третьими сторонами.

Эффективность

В узком смысле применительно к распределённым системам под эффективностью будет пониматься минимизация накладных расходов, связанных с распределённым характером системы. Поскольку эффективность в данном узком смысле может противоречить безопасности, открытости и надёжности системы, следует отметить, что требование эффективности в данном контексте является наименее приоритетным. Например, на поддержку логической целостности данных в распределённой системе могут тратиться значительные ресурсы времени и памяти, однако система с недостоверными данными вряд ли нужна пользователям.

Желательным свойством промежуточной среды является возможность организации эффективного обмена данными, если взаимодействующие программные компоненты находятся на одном компьютере. Эффективная промежуточная среда должна иметь возможность организации их взаимодействия без затрагивания стека TCP/IP. Для этого могут использоваться системные сокеты (unix sockets) в POSIX системах или именованные каналы (named pipes).

Устойчивость распределённой системы связана с понятием **масштабируемости**, но не эквивалентна ему. Допустим, система использует набор серверов, обрабатывающих запросы, и один диспетчер запросов, который распределяет запросы пользователей между серверами. Такая система может считаться достаточно хорошо масштабируемой, однако диспетчер является уязвимой точкой такой системы. С другой стороны, система с единственным

сервером может быть устойчива, если существует механизм его автоматической замены в случае выхода его из строя, однако она вряд ли относится к классу хорошо масштабируемых систем. На практике достаточно часто встречаются распределённые системы, не удовлетворяющие данным требованиям: например, любая система с уникальным сервером БД, реализованным в виде единственного компьютера, имеет уникальную точку сбоя. Выполнение требований устойчивости и масштабируемости обычно связано с некоторыми дополнительными расходами, что на практике может быть не всегда целесообразно. Однако используемые при построении распределённых систем технологии должны допускать принципиальную возможность создания устойчивых и высоко масштабируемых систем.

Заблуждения о распределённых системах:

Заблуждение №1. Сеть надёжна

И разработчики, и архитекторы полагают, что сеть надёжна, но это не так. Хотя со временем сети стали надёжнее, факт в том, что они всё ещё остаются в целом ненадежными. Это важно для всех распределённых архитектур, потому что все стили распределённой архитектуры полагаются на сеть для связи с сервисами и между сервисами. Чем больше система зависит от сети, тем потенциально менее надёжной она становится.

Заблуждение №2. Задержки равны нулю

В отличие от взаимодействия в рамках одного процесса операционной системы, отправка сообщений по сети требует намного больше времени, что необходимо учитывать при проектировании сервисов чувствительных к задержкам.

Если предположить, что в среднем на один запрос приходится 100 миллисекунд задержки, то объединение 10 вызовов служб для выполнения определённой бизнес-функции добавляет к запросу 1 000 миллисекунд! Знать среднее значение задержки очень важно, но ещё важнее знать 95-99-й процентили. Если средняя задержка может составлять всего 60 миллисекунд, то 95-й процентиль может составлять 400 миллисекунд! Обычно именно эта «длинная хвостовая» задержка убивает производительность в распределённой архитектуре. В большинстве случаев архитекторы могут получить значения задержки от сетевого администратора.

Заблуждение №3. Пропускная способность бесконечна

Пропускная способность обычно не является проблемой в монолитных архитектурах, поскольку, как только обработка идёт в монолите, для обработки бизнес-запроса не требуется или почти не требуется пропускная способность. Когда системы разбиваются на более мелкие единицы развёртывания (сервисы) в распределённой архитектуре, связь с этими сервисами и между ними значительно использует пропускную способность, что приводит к замедлению работы сети, тем самым влияя на задержку и надёжность.

Чтобы проиллюстрировать важность этого заблуждения, рассмотрим две службы, показанные на рисунке. Предположим, что служба A управляет списками желаний на сайте, а служба B — профилем клиента. Когда запрос на список пожеланий поступает в службу A, она должна сделать межсервисный вызов в службу B профиля клиента, чтобы получить имя клиента, потому что эти данные необходимы в контракте ответа для списка пожеланий, но служба списка пожеланий на стороне A не имеет этого имени. Служба профиля клиента возвращает 45 атрибутов общим объёмом 500 кб службе списка пожеланий, которой нужно только имя (200 байт). Запросы на элементы списка пожеланий происходят примерно 2 000 раз в секунду. Это означает, что межсервисный вызов от службы списка пожеланий к службе профиля клиента происходит 2 000 раз в секунду. При 500 кб на каждый запрос объём полосы пропускания, используемой для этого одного межсервисного вызова (из сотен, выполняемых в эту секунду), составляет 1 Гб.

Заблуждение №4. Сеть безопасна (защищена)

Безопасность становится намного сложнее в распределённой архитектуре. Каждая конечная точка каждой распределённой единицы развёртывания должна быть защищена, чтобы неизвестные или плохие запросы не попали в эту службу. Площадь поверхности для угроз и атак увеличивается в разы при переходе от монолитной к распределённой архитектуре. Необходимость защищать каждую конечную точку даже при межсервисном взаимодействии — ещё одна причина того, что производительность, как правило, ниже в синхронных, высокораспределённых архитектурах, таких как микросервисы или архитектуры на основе сервисов.

Заблуждение №5. Топология никогда не меняется

Это заблуждение относится к общей топологии сети, включая все маршрутизаторы, концентраторы, коммутаторы, брандмауэры, сети и устройства, используемые в общей сети. Архитекторы предполагают, что топология никогда

не меняется. Но чаще всего топология меняется, что может приводить к неустойчивости сети.

Заблуждение №6. Администратор всегда один

Если сеть достаточно большая, то, как правило, обслуживанием её занимаются несколько администраторов. Данное заблуждение следует учитывать при сборе сведений о сети.

Заблуждение №7. Каналы связи ничего не стоят

Иногда архитекторы ошибочно полагают, что необходимая инфраструктура уже создана и достаточна для выполнения простейшего REST-запроса или разделения монолитного приложения. распределённые архитектуры обходятся значительно дороже монолитных, в первую очередь из-за увеличения потребностей в дополнительном оборудовании, серверах, шлюзах, брандмауэрах, новых подсетях, прокси-серверах и так далее.

Приступая к созданию распределённой архитектуры, следует проанализировать текущую топологию сети с точки зрения ёмкости, пропускной способности, задержек и зон безопасности. Конечно, часть этих вопросов может быть решена, если система развёртывается в облачной инфраструктуре.

Заблуждение №8. Сеть гомогенна

В реальности сеть редко строится на оборудовании и технологиях одного производителя. Это может приводить к тому, что оборудование одного производителя плохо интегрируется с оборудованием другого производителя. Как и с заблуждением №7 следует учитывать, что при использовании облачного окружения данной проблемой можно пренебречь.

Помимо восьми заблуждений распределённых вычислений, описанных ранее, существуют и другие вопросы и проблемы распределённой архитектуры, которые отсутствуют в монолитных архитектурах.

4. Паттерны доступа к данным

4.1. Data Access Object (DAO)

Способ доступа к данным бывает разным и зависит от источника данных. Способ доступа к персистентному хранилищу, например, к базе данных, очень зависит от типа этого хранилища (реляционные базы данных, объектно-ориентированные базы данных, однородные или «плоские» файлы и т.д.) и от конкретной реализации хранилища.

Обычно приложения совместно используют распределённые компоненты для представления персистентных данных, например, компоненты управления данными. Считается, что приложение использует управляемую компонентом персистенцию для своих компонентов управления данными, если эти компоненты явно обращаются к персистентным данным, — то есть компонент содержит код прямого доступа к хранилищу данных. Приложение с более простыми требованиями может вместо компонентов управления данными использовать сессионные компоненты или сервлеты с прямым доступом к хранилищу данных для извлечения и изменения данных. Также приложение могло бы использовать компоненты управления данными с управляемой контейнером персистенцией, передавая, таким образом, контейнеру функции управления транзакциями и деталями персистенции.

DAO реализует необходимый для работы с источником данных механизм доступа. Источником данных может быть персистентное хранилище (например, RDBMS), внешняя служба (например, B2B-биржа), репозиторий (LDAP-база данных), или бизнес-служба. Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов.

Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн позволяет DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO выполняет функцию адаптера между компонентом и источником данных.

На рисунке изображена схема шаблона DAO.

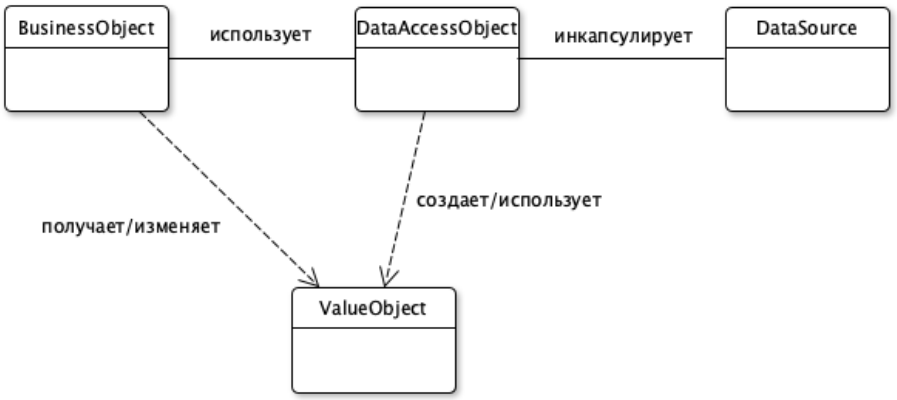
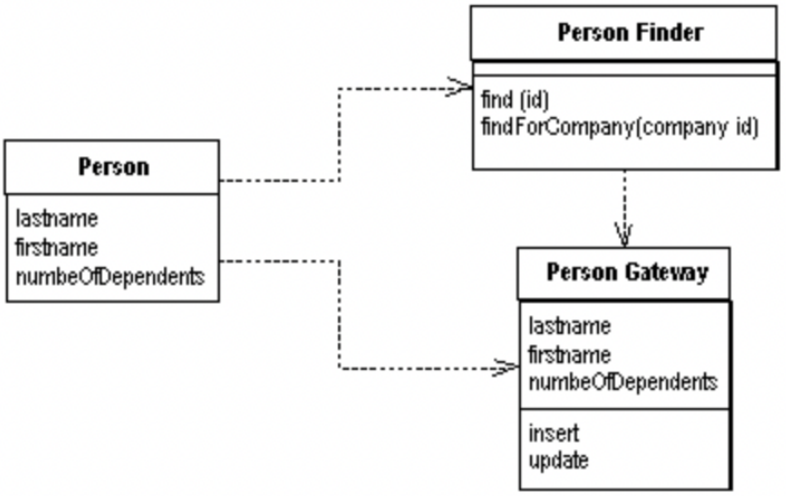


Схема шаблона DAO

4.2. Row Data Gateway

Шлюз к данным таблицы. Объект выступает в роли шлюза к отдельной записи в источнике данных. Один экземпляр на одну запись.



Пример паттерна Row Data Gateway

Встраивание кода доступа к БД в объекты, хранящиеся в памяти, может привести в некоторым неудобствам. Если объекты содержат бизнес-логику, добавление взаимодействия с БД сломает разделение на слои и увеличит сложность. Модульное тестирование станет неудобным, если объекты, хранящиеся в памяти, жёстко привязаны к БД. Тесты станут медленнее из-за доступа к БД. Объект шлюза к записи представляется в точности, как запись в

БД, но при этом даёт возможность доступа посредством штатных средств языка программирования. Все подробности доступа к БД скрыты за этим интерфейсом.

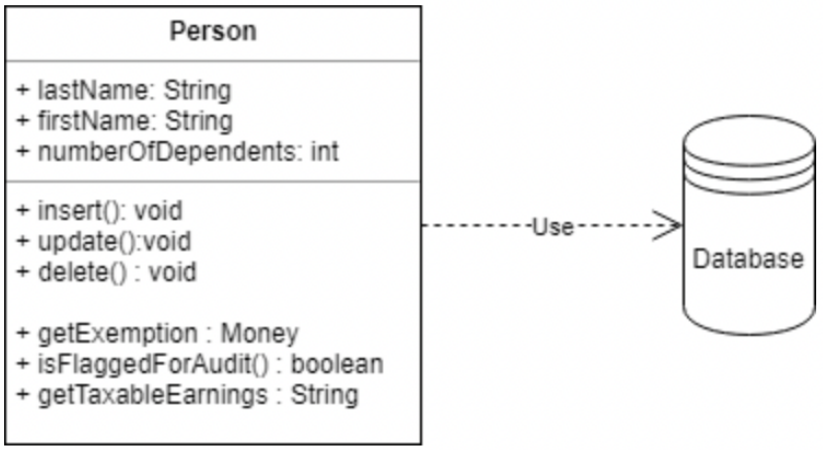
Шлюз выступает в роли интерфейса к строке данных и прекрасно подходит для применения в сценариях транзакции (Transaction Script). Реализация шлюза записи данных включает в себя только логику доступа к базе данных и никакой логики домена. Добавление логики домена в шлюз записи данных превращает его (шлюз) в активную запись (Active Record).

Пример на рисунке выше: объект шлюза PersonGateway, данные о записи из таблицы person и методы insert и update, которые позволяют прозрачно работать с записью.

4.3. Active Record

Активная запись — Один объект управляет и данными, и поведением.

Объект, выполняющий роль оболочки для строки таблицы или представления базы данных. Он инкапсулирует доступ к базе данных и добавляет к данным логику домена. В основе типового решения активная запись лежит модель предметной области (Domain Model). Каждая активная запись отвечает за сохранение и загрузку информации в базу данных, а также за логику домена, применяемую к данным. Это может быть вся бизнес-логика приложения. Впрочем, иногда некоторые фрагменты логики домена содержатся в сценариях транзакции (Transaction Script), а общие элементы кода, ориентированные на работу с данными — в активной записи.



Пример реализации паттерна Active Record

Активная запись очень похожа на шлюз записи данных (Row Data Gateway). Принципиальное отличие между ними состоит в том, что шлюз записи данных содержит только логику доступа к базе данных, в то время как активная запись содержит и логику доступа к данным, и логику домена. Активная запись хорошо подходит для реализации базовой логики домена, в частности, операций создания, считывания, обновления и удаления. Кроме того, она прекрасно справляется с извлечением и проверкой на правильность отдельной записи.

Пример, представленный на рисунке выше: объект "Person" содержит данные об одной персоне и методы: добавить, обновить или удалить. Паттерн продвигает идею сосредоточения модели и логики в одном классе.

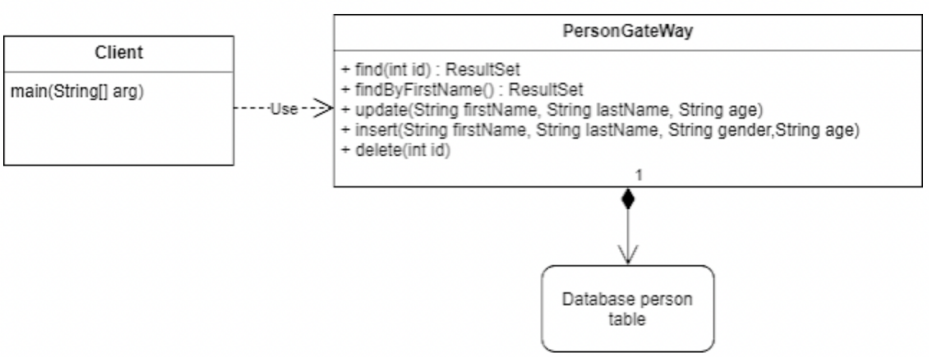
4.4. Table Data Gateway

Шлюз данных к таблице. Объект выступает в качестве шлюза между данными в приложении и в БД. Один объект работает сразу со всеми записями в таблице.

Шлюз табличных данных имеет простой интерфейс, обычно состоящий из нескольких методов поиска для получения данных из базы данных и методов обновления, вставки и удаления. Каждый метод отображает входные параметры в вызов SQL и выполняет SQL для соединения с базой данных. Шлюз табличных данных, как правило, не имеет состояния, поскольку его роль заключается в передаче данных с базы в бизнес-логику и обратно.

Таким образом, Table Data Gateway паттерн — это прослойка, которая отделяет SQL от бизнес-логики. Методы шлюза таблицы данных используются другими объектами для взаимодействия с базой данных. Как правило, для каждой таблицы базы данных создаётся собственный шлюз таблицы данных.

Это наиболее простое типовое решение интерфейса базы данных, поскольку оно замечательно отображает таблицы или записи баз данных на объекты. Одно из преимуществ использования шлюза таблицы данных для инкапсуляции доступа к базе данных состоит в том, что этот интерфейс может применяться и для обращения к базе данных с помощью средств языка SQL, и для работы с хранимыми процедурами. Данное решение подходит практически для любой платформы, так как представляет собой не более чем оболочку для операторов SQL.

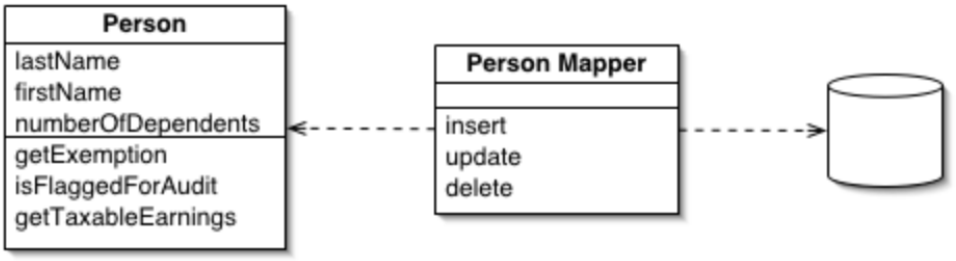


Пример реализации паттерна Table Data Gateway

Пример на рисунке выше : объект шлюза PersonGateway содержит методы для доступа к таблице person в БД. Методы содержат SQL-код для выборки, вставки, обновления и удаления. Объект может содержать специальную выборку, например, поиск по компании.

4.5. Data Mapper

Преобразователь данных. Программная прослойка, разделяющая объект и БД. Его обязанность — пересылать данные между ними и изолировать их друг от друга.



Пример реализации паттерна Data Mapper

Это слой программного обеспечения, которое отделяет объекты, расположенные в оперативной памяти, от базы данных. В функции преобразователя данных входит передача данных между объектами и базой данных и изоляция их друг от друга. В результате этого объекты, расположенные в оперативной памяти, могут даже не «подозревать» о самом факте присутствия

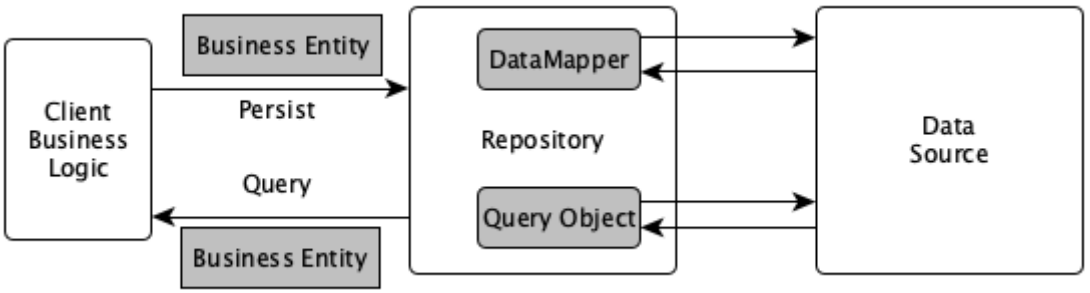
базы данных. Им не нужен SQL-интерфейс и тем более схема базы данных. Поскольку преобразователь данных является разновидностью преобразователя «Адаптер», он полностью скрыт от уровня домена.

В большинстве случаев преобразователь данных применяется для того, чтобы схема базы данных и объектная модель могли изменяться независимо друг от друга. Как правило, подобная необходимость возникает при использовании модели предметной области (Domain Model). Основным преимуществом преобразователя данных является возможность работы с моделью предметной области без учёта структуры базы данных как в процессе проектирования, так и во время сборки и тестирования проекта. В этом случае объектам домена ничего не известно о структуре базы данных, поскольку все отображения выполняются преобразователями.

4.6. Repository (Репозиторий)

Одним из наиболее часто используемых паттернов при работе с данными является паттерн Репозиторий. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Репозиторий — это фасад для доступа к базе данных. На рисунке 15 изображена схема паттерна Репозиторий.



Пример схемы паттерна репозиторий

Весь код приложения за пределами репозитория работает с базой данных через него и только через него. Таким образом, репозиторий инкапсулирует в себе логику работы с базой данных, это, в некотором смысле, слой объектно-реляционного отображения в приложении. Более точно, репозиторий,

или хранилище, это интерфейс для доступа к данным одного типа — одной доменной сущности. В простейшем случае это означает и одну таблицу реляционной базы данных. Доступ к данным организуется через совокупность всех репозиториев.

Система со сложной моделью области определения может быть упрощена при помощи дополнительного уровня, например Data Mapper, который бы изолировал объекты от кода доступа к БД. В таких системах может быть полезным добавление ещё одного слоя абстракции поверх слоя распределения данных, в котором бы был собран код создания запросов. Это становится более важным, когда в области определения множество классов или при сложных, тяжёлых запросах.

Паттерн Repository посредничает между слоем бизнес-логики и сущностей домена и слоем распределения данных, работая, как обычная коллекция объектов сущности домена. Объекты-клиенты создают описание запроса декларативно и направляют их к объекту-репозиторию (Repository) для обработки. Объекты могут быть добавлены или удалены из репозитория, как будто они формируют простую коллекцию объектов. А код распределения данных, скрытый в объекте Repository, позаботится о соответствующих операциях в базе данных / хранилище данных незаметно для разработчика.

В двух словах, паттерн Repository инкапсулирует объекты, представленные в хранилище данных и операции, производимые над ними, предоставляя более объектно-ориентированное представление реальных данных. Repository также преследует цель достижения полного разделения и односторонней зависимости между уровнями области определения и распределения данных.

Далее мы рассмотрим другие часто используемые паттерны.

4.7. Domain Model (Модель предметной области)

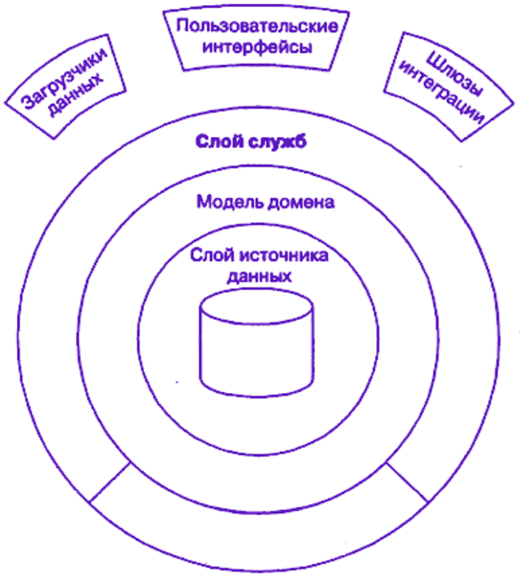
Паттерн заключается в использовании объектной модели, каждый класс которой представляет некую наделённую смыслом сущность. Можно выделить два вида моделей предметной области. В простой модели один объект домена, как правило, соответствует одной таблице из базы данных. В сложной модели возможно также использование наследования, создание иерархии объектов.

Одно из ключевых правил при использовании данного паттерна заключается в том, что модель предметной области должна быть наиболее независима от других слоёв системы.

4.8. Service Layer (Слой служб)

Идея паттерна заключается в создании дополнительного слоя, который содержит в себе типовые действия при использовании бизнес-логики. Базовый вариант реализации слоя служб предполагает создание интерфейса доступа к домену. Слой служб это набор интерфейсов, которые не содержат никакой бизнес-логики, а работают только с моделью предметной области.

Пример разделения представлен на рисунке. Слой служб обеспечивает доступ к отдельным компонентам приложения, также в качестве службы может являться отделённый интерфейс к удалённому сервису. Слой служб скрывает конкретный способ реализации от домена.



Пример разделения на слои

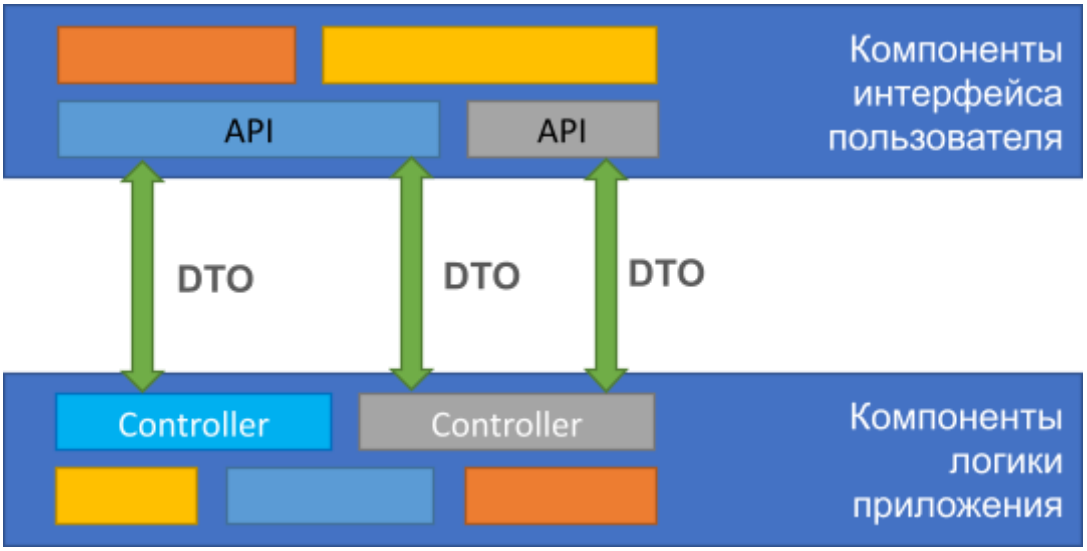
4.9. Data Transfer Object (Объект Передачи Данных)

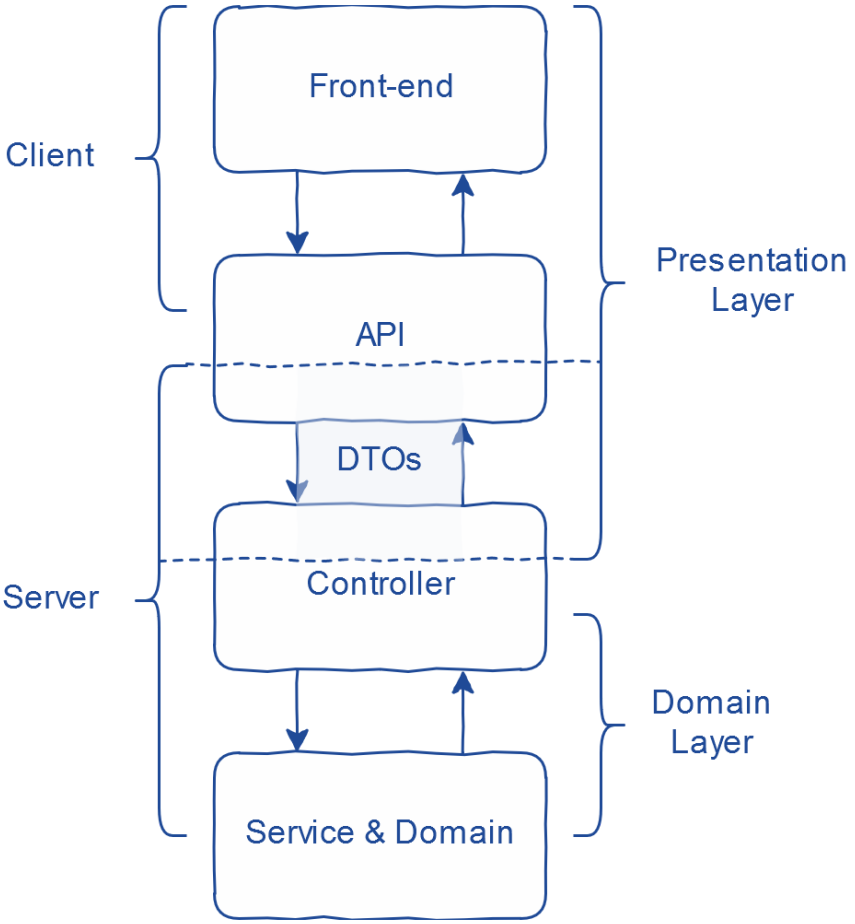
Data Transfer Object — это объект, который пересылает данные между процессами для уменьшения количества вызовов методов.

Что такое DTO (Data Transfer Object)?

Зачастую в клиент-серверных приложениях данные на клиенте (*слой представления*) и на сервере (*слой предметной области*) структурируются по-разному. На стороне сервера это позволяет комфортно хранить данные в БД или оптимизировать использование данных в угоду производительности, при этом отображать данные на клиенте в манере *“user-friendly”*. Для серверной части нужно найти способ, как переводить данные из одного формата в другой. Конечно, существуют и другие архитектуры приложений, но мы остановимся на текущей ради упрощения. DTO-подобные объекты могут использоваться между любыми двумя слоями представления данных.

DTO — это так называемый value-object на стороне сервера, который хранит данные, используемые в слое представления. Мы разделим DTO на те, что мы используем при запросе (*Request*), и те, что мы возвращаем в качестве ответа сервера (*Response*).

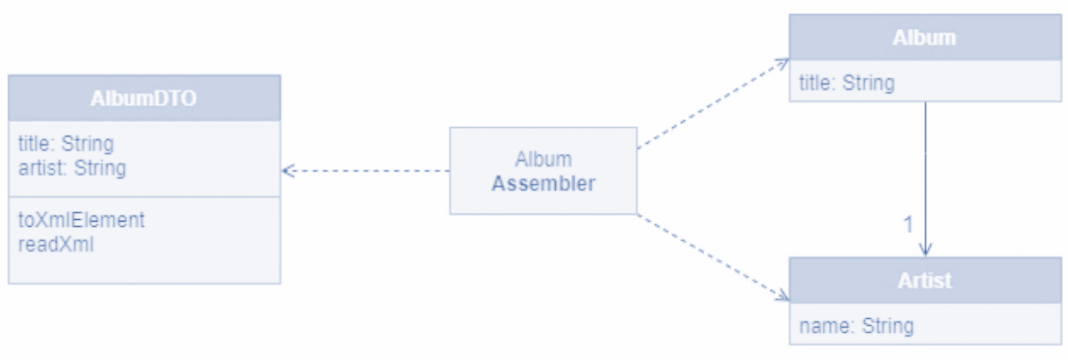




Проблема: При работе с любым интерфейсом получения данных каждый запрос может быть достаточно затратен. Это ведёт к попыткам уменьшить количество вызовов и к необходимости передавать больше данных за единственный вызов. Чтобы реализовать подобное поведение, можно использовать множество параметров вызова, однако это может привести к появлению неэффективного кода. Также подобное решение невозможно в тех языках, в которых из метода может быть возвращено лишь одно значение.

Решение. Паттерн Data Transfer Object может хранить всю необходимую для вызова информацию.

Особенности. На рисунке представлен схематический пример использования рассматриваемого паттерна.



Пример использования паттерна Data Transfer Object

Реализуемый паттерн должен быть сериализуем для удобства передачи по сети.

Данный паттерн применим во всех случаях, когда необходимо передать несколько элементов данных между двумя процессами в одном вызове метода.

Также стоит отметить, что существует несколько альтернатив использованию Data Transfer Object.

Первая альтернатива — отказ от использования объекта и использование метода настройки со многими аргументами или использование метода получения с несколькими возвращаемыми аргументами. Проблема подобного подхода заключается в том, что некоторые языки программирования (например, Java) допускают лишь единственный аргумент в качестве возвращаемого из метода значения. В таком случае могут возникнуть проблемы с получением данных.

Вторая альтернатива — использование некоторой формы строкового представления напрямую, без использования объекта, играющего роль интерфейса к ней. Проблема в данном случае заключается в том, что все остальные объекты системы также становятся связанными со строковыми представлениями и в случае, если потребуется, например, изменить XML на JSON, эти изменения могут повлечь полную переработку системы с целью добавления поддержки нового формата.

Пример использования:

Часто в клиент-серверных приложениях данные в слое представления и в слое предметной области структурируются по-разному. На стороне сервера подобное решение предоставляет возможность удобного хранения данных в базе данных или оптимизации их использования при работе бизнес-логики. В то же время требуется предоставить возможность отображения данных для слоя клиента, то есть перевести данные из одного формата в другой. Data Transfer

Object, расположенный между двумя этими слоями, способен организовать подобное взаимодействие в системе.

Сводная таблица по рассмотренным паттернам

Типовым решением для реализации доступа к данным будет паттерн Репозиторий, интерфейс которого — работа с коллекциями сущностей домена, а внутренняя реализация зависит от используемой базы данных и нормализации данных на уровне представления в конкретной СУБД. Ниже представлена таблица 4 по рассмотренным паттернам.

Сводная таблица

Название	Row Data Gateway	Active Record	Table Data Gateway	Data Mapper	Repository
Доступ	Одна запись в БД	БД и Домен	Одна таблица в БД	БД	БД
SOLID	Да	Не совсем (может не соответствовать Single Responsibility)	Да	Да	Не совсем (может не соответствовать Single Responsibility)
Запросы	SQL	SQL	SQL	Любого вида	Любого вида
Тип шаблона	Поведенческий	Структурный	Поведенческий	Структурный	Поведенческий

Глоссарий

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объёмов.

Бизнес-правила — совокупность правил, принципов, зависимостей поведения объектов предметной области. Последнее представляет собой область человеческой деятельности, которую система поддерживает.

Компонент — множество классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Пользовательский интерфейс — интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Data Transfer Object (DTO) — один из шаблонов проектирования, используется для передачи данных между подсистемами приложения. Data Transfer Object, в отличие от Business Object или Data Access Object не должен содержать какого-либо поведения.

Object-Relational Mapping (ORM) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Существуют как проприетарные, так и свободные реализации этой технологии.

Model-View-Controller (MVC) — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

Model-View-Presenter (MVP) — шаблон проектирования, производный от MVC, который используется в основном для построения пользовательского интерфейса. Элемент Presenter в данном шаблоне берёт на себя функциональность посредника (аналогично контроллеру в MVC) и отвечает за управление событиями пользовательского интерфейса (например, использование мыши) так же, как в других шаблонах обычно отвечает представление.

Model-View-ViewModel (MVVM) — шаблон проектирования архитектуры приложения. Представлен в 2005 году Джоном Госсманом как модификация шаблона Presentation Model. Ориентирован на современные платформы разработки. MVVM удобно использовать вместо классического MVC и ему подобных в тех случаях, когда в платформе, на которой ведётся разработка, есть «связывание данных». В шаблонах проектирования MVC/MVP изменения в пользовательском интерфейсе не влияют непосредственно на Модель, а предварительно идут через Контроллер (англ. Controller) или Presenter. В таких технологиях как WPF и Silverlight есть концепция «связывания данных», позволяющая связывать данные с визуальными элементами в обе стороны. Следовательно, при использовании этого приёма применение модели MVC становится крайне неудобным из-за того, что привязка данных к представлению напрямую не укладывается в концепцию MVC/MVP.

Дополнительные материалы

1. Выступление Jimmy Bogard на тему Solid Architecture in Slices not Layers.
2. Статья Component Based Architecture.

Используемые источники

1. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения.
2. Диаграмма компонентов.
3. Контрактное программирование: в Википедии, на Хабре.
4. UML-диаграммы.
5. The Clean Architecture (дата обращения: 06.06.2021).
6. Мартин, Р. Чистый код: создание, анализ и рефакторинг. / Р. Мартин. — Москва: Юпитер, 2018. — 464 с.
7. Фаулер М. Архитектура корпоративных приложений.: Пер. с англ. — М.: Издательский дом «Вильямс», 2006. — 544 с.
8. С. Макконнелл (Steve McConnell, Code Complete). Совершенный код.