

Лекция 9: Архитектурные стили

Юрий Литвинов

yurii.litvinov@gmail.com

04.04.2022

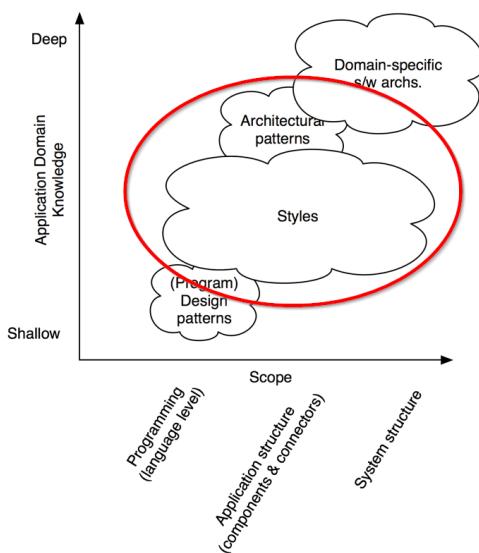
1. Архитектурные шаблоны и стили

Эта лекция — пожалуй, самая «архитектурная» в этом курсе, на ней пойдёт речь о наиболее известных архитектурных стилях. Вообще, архитектурный стиль — это набор решений, которые:

1. применимы в выбранном контексте разработки,
2. задают ограничения на принимаемые архитектурные решения, специфичные для определённых систем в этом контексте,
3. приводят к желаемым положительным качествам получаемой системы.

Есть ещё архитектурные шаблоны — это именованный набор ключевых проектных решений по эффективной организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях.

Определения довольно размыты, но суть дела поясняет рисунок:



Паттерны проектирования, которые обсуждались в предыдущих лекциях — самые «тактические» элементы архитектуры. Они никак не привязаны к предметной области и появляются на уровне реализации небольших подсистем или даже конкретных классов. Архитектурные стили, о которых в основном пойдёт речь сегодня, применимы уже не для всех проектов вообще, а в предпочтительных для каждого стиля предметных областях — одни стили хорошо работают во встроенных системах, другие — сетевых приложениях, третьи — в информационных системах (отсюда «применимы в выбранном контексте разработки» из определения). И решения, диктуемые стилями, применяются не на уровне конкретных классов, а на уровне подсистем или даже целой системы.

Архитектурные шаблоны — это более специализированная вещь, чем стили, и несколько более «тактическая» (хотя и не настолько, как паттерны). Архитектурные шаблоны диктуют типовые решения для типовых задач, например, организация системы в виде тройки Sense-Compute-Control в робототехнике, или Model-View-Controller в пользовательских интерфейсах. Model-View-Controller не претендует на то, чтобы диктовать архитектуру всего приложения, и тем отличается от архитектурных стилей — обычно MVC лишь вершина айсберга, ответственная за общение с пользователем, а настоящая Архитектура начинается на уровне бизнес-логики, с которым работает Model.

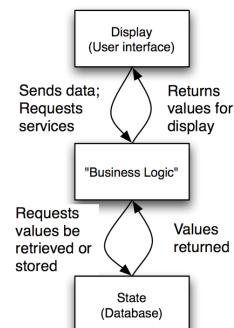
Ещё выше по масштабности и глубже по погружению в предметную область находятся предметно-ориентированные архитектуры или Reference Architectures (например, Space AVionics Open Interface aRchitecture от ESA, Connected Vehicle Reference Implementation Architecture и т.д. и т.п.). В каждой предметной области они свои и, как правило, даже диктуются стандартами. Рассматривать их в этом курсе мы не будем в силу их чрезмерной специфичности.

2. Архитектурные шаблоны

Вот несколько примеров архитектурных шаблонов. Они тоже специфичны для предметных областей, но некоторые часто встречающиеся достойны рассмотрения.

2.1. State-Logic-Display

State-Logic-Display, также известный как «трёхзвенная архитектура», часто рассматривается как архитектурный стиль, часто — как архитектурный шаблон, так что вообще разделение на архитектурные шаблоны и архитектурные стили весьма условно. Многие приложения могут целиком быть реализованы по трёхзвенной схеме, причём ни в одном из этих звеньев не будет содержательной архитектурной сложности, тогда это что-то вроде стиля. Если трёхвенка — это только способ решения одной из задач, то это архитектурный шаблон. Вообще, трёхвенка предполагает разделение приложения на часть, отвечающую за представление и взаимодействие с пользователем (и ничего больше), бизнес-логику и слой хранения данных.



Наличие чёткого разделения ответственности между слоями делает каждый из них управляемым, а функциональность — переиспользуемой. Например, пользовательский интерфейс легко поменять, сделав вместо десктопного приложения мобильное, не меняя бизнес-логики. В больших приложениях все три части могут быть физически размещены на разных машинах, причём пользовательских клиентов может быть много и они все работают со слоем бизнес-логики, который один. Или можно иметь несколько серверов с бизнес-логикой, лишь бы они смотрели на одну базу данных. Которая тоже может быть распределённой, так что это очень хорошо сказывается на масштабировании.

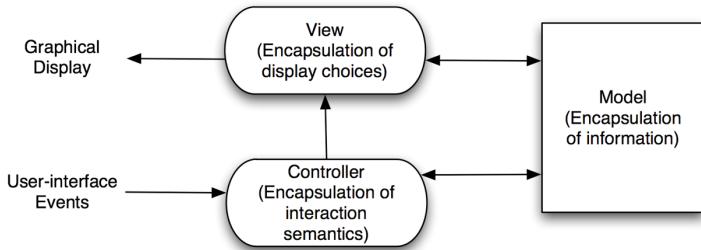
Главные архитектурные ограничения — клиент не знает ничего о БД и не может с ней напрямую взаимодействовать, бизнес-логика сама не пытается хранить информацию и не содержит ничего, что касается взаимодействия с пользователем, база данных не пытается делать ничего нетривиального с данными.

Приложения, устроенные таким образом — это подавляющее большинство веб-приложений и информационных систем (которые чаще всего сами веб-приложения), многопользовательские игры. В них клиент может быть очень продвинутым и очень сложным, но он не вправе заниматься реализацией игровой логики и не может хранить данные, кроме тех, что относятся к взаимодействию с пользователем (например, его логин).

2.2. Model-View-Controller

Model-View-Controller — это архитектурный шаблон, который иногда классифицируют как паттерн проектирования: собственно, в книжке про паттерны Эриха Гамма со товарищи с него и начинается изложение. Так что даже разделение на паттерны и архитектурные шаблоны тоже весьма условно. Кажется, что это всё-таки что-то большее, чем паттерн, поскольку он не предписывает наличия конкретных классов, да и сам часто реализуется через некоторые паттерны (например, «Наблюдатель», «Команда»).

Архитектурный шаблон устроен следующим образом:



© N. Medvidovic

View отвечает за отображение данных пользователю и только за это. Пользовательский ввод поступает в **Controller**, ответственность которого — обеспечить логику взаимодействия с пользователем и при необходимости управлять для этого **View**. **Model** — компонент, хранящий в себе все данные и, как правило, включающий в себя также и бизнес-логику приложения. Контроллер обрабатывает пользовательский ввод, сообщает о требуемых действиях модели, модель их выполняет, меняет данные и рассыпает нотификацию о том, что в ней что-то изменилось. Эту нотификацию получает представление, читает информацию из модели и обновляет себя.

Архитектурные ограничения: представление может только читать из модели, команды модели может отдавать только контроллер. Сигнал об обновлениях модель рассыпает сама, что позволяет иметь несколько разных представлений, отображающих одну модель, которые будут синхронно обновляться. Контроллер — единственное место системы, через которое проходят все команды пользователя.

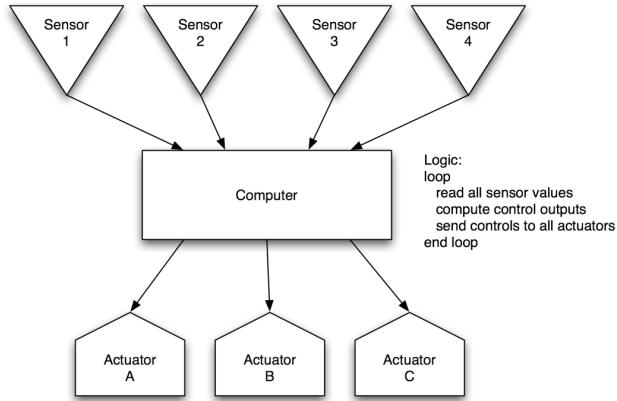
Чем это хорошо — так же, как в трёхзвенке, имеется чёткое разделение ответственности между компонентами и возможность выкинуть представление (или контроллер) и заменить на новое. Кроме того, контроллер — это естественное место для реализации функциональности Undo/Redo и обычно реализуется с помощью паттерна «Команда».

Model-View-Controller отличается от трёхвенке тем, что не специфицирует, кому где хранить данные (поэтому применим в приложениях, которым данные особо хранить не надо), но требует наличия выделенного элемента, отвечающего за обработку пользовательского ввода (в трёхвенке это делал **Display**, вместе с выводом информации).

Типичные приложения, построенные по такому принципу — это большинство десктопных приложений с развитым пользовательским интерфейсом. Как правило, впрочем, Model-View-Controller является лишь малой частью их архитектуры, а вся логика и вся архитектурная сложность скрыта под **Model**.

2.3. Sense-Compute-Control

Sense-Compute-Control — архитектурный шаблон, применяющийся прежде всего в робототехнике и схожих областях (например, «умных домах»). Он предполагает разделение работы системы на три фазы — снятие показания с датчиков, вычисление управляющего воздействия, посылка управляющего воздействия на актуаторы:



© N. Medvidovic

При этом вся система работает в бесконечном цикле, попеременно выполняя эти три фазы.

Очень простые системы (например, робот, едущий по датчику расстояния вдоль стеки) на самом деле больше ничего интересного не содержат, но в общем случае фаза Compute может включать в себя построение и обновление модели внешнего мира (например, целый большой и страшный SLAM¹), кучу сложной логики по определению поведения исходя из текущей модели мира и новых данных сенсоров.

Хорошо это тем, что чётко определяет архитектуру системы и для простых систем фактически решает все архитектурные вопросы. Для сложных систем это может быть только самый внешний каркас процесса работы, но поэтому это и не архитектурный стиль, а всего лишь шаблон.

3. Архитектурные стили

Архитектурные стили менее специализированы, чем архитектурные шаблоны. Так же, как и шаблоны, стили имеют имя и набор известных свойств, которые позволяют выбрать стиль, подходящий под решение задачи. Архитектурные стили в разработке ПО часто сравнивают с архитектурными стилями в архитектуре. Так же, как в архитектуре, стили могут быть совсем разными, при этом решать схожие задачи и, в конечном итоге, быть отражением вкусовых предпочтений архитектора.



© N. Medvidovic

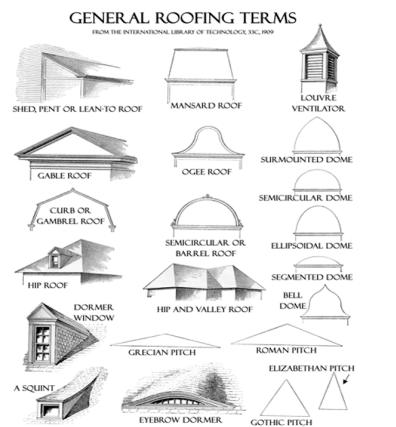
¹ Simultaneous Localization And Mapping

Однако не все архитектурные стили хороши для всех ситуаций. Пример, который приводил N. Medvidovic в своей лекции — в Калифорнии делают лёгкие каркасные дома из деревянного бруса, а в Сербии, откуда он родом — большие дома из камня, передающиеся из поколения в поколение. Дело в том, что в Калифорнии высока сейсмическая активность, которая деревянным домам не страшна, а каменные бы и пары лет неостояли. В Сербии землетрясений не бывает, а камня в достатке.

При этом одна система вполне может включать в себя несколько разных архитектурных стилей (опять-таки, по аналогии со строительством, где есть, например, разные архитектурные стили крыши). Для каждой подсистемы стиль может быть свой. Например, система может быть организована в слоистом стиле, но слой бизнес-логики реализован в стиле Pipes and Filters, а слой пользовательского интерфейса — в стиле «ядро и плагины». Тем не менее, обычно прослеживается некий «главный» стиль — высокоравневая структура системы, которая и связывает компоненты, потенциально написанные очень по-разному.

Архитектурные стили используются, чтобы получить следующие преимущества.

- Переиспользование архитектуры. Создание архитектуры может быть сложной задачей, особенно когда у вас есть только требования и «чистый лист». Выбор стиля сужает пространство решений и направляет архитектурную мысль. При этом для новых задач можно применять хорошо известные и изученные решения, обладающие известными достоинствами и недостатками применительно к вашей ситуации.
- Переиспользование кода. Часто у архитектурных стилей бывают неизменяемые части, которые можно один раз реализовать, а затем переиспользовать в каждой системе. Трёхзвенная архитектура, например, реализуется многими библиотеками для разработки веб-приложений, для событийно-ориентированных стилей есть хороший middleware (например, уже упоминавшийся ROS), для распределённых стилей — технологии наподобие gRPC или WCF, и т.д. Это существенно сокращает затраты на разработку.
- Упрощение общения и понимания системы. Как и в случае с паттернами, достаточно просто назвать стиль по имени и не надо объяснять, как что устроено — опытные разработчики вас сразу поймут.
- Упрощение интеграции приложений, на тактическом уровне за счёт переиспользования стандартов и middleware, типичных для стиля, на стратегическом — за счёт того, что понятно, куда и как встраиваться, не нарушив архитектурные ограничения каждого из приложений.
- Применение специфичных для стиля методов анализа. Поскольку стили накладывают ограничения на структуру систем, иногда эти ограничения достаточно жёсткие, чтобы про систему можно было что-то доказать или что-то посчитать. Хороший



© N. Medvidovic

пример в этом плане — стиль Pipes and Filters, к которому применимы алгоритмы анализа графов, которые можно использовать для анализа пропускной способности системы, её надёжности и т.п.

- Специфичные для стиля методы визуализации — тот же Pipes and Filters удобно рисовать с помощью визуальных языков, ориентированных на данные, таких как Data Flow Diagram. Или становится возможным использование предметно-ориентированных языков — например, на том же Pipes and Filters построено программирование в LabVIEW и Matlab/Simulink. Pipes and Filters тут не исключение, например, Microsoft Robotics Developer Studio имела по сути визуальный DSL для создания программ в распределённом веб-сервисном стиле.

Все стили фиксируют ограничения на возможные архитектуры, и какие именно эти ограничения — и есть описание стиля. Стили определяются тремя основными соображениями: набор используемых в стиле элементов, набор правил, по которым эти элементы соединяются, и семантика, стоящая за элементами. Элементы могут быть компонентами, соединителями, элементами данных и т.п., например, некоторые стили описывают объекты и вызовы методов, некоторые — сервера и каналы связи, некоторые — фильтры и каналы данных. Правила соединения элементов — это набор «топологических» ограничений на то, кто с кем может соединяться, и это, как правило, и есть самая суть стиля. Например, строгий слоистый стиль требует, чтобы элементы одного слоя могли общаться только друг с другом и с элементами слоя ниже. Семантика, стоящая за элементами, ограничивает их возможности, например, фильтрам можно запретить иметь собственное состояние, а каналам — преобразовывать данные.

3.1. Объектно-ориентированный стиль

«Сырой» объектно-ориентированный стиль — когда программа представляется в виде набора взаимодействующих объектов в обычном объектно-ориентированном стиле. Компонентами в этом стиле выступают объекты, соединителями — вызовы методов, ограничения — объекты полностью контролировать своё внутреннее состояние и менять его можно только через вызовы методов, при этом реализация этих методов должна быть полностью скрыта от других объектов. В общем, обычные требования ООП.

Я бы вообще не называл объектно-ориентированный стиль архитектурным стилем, поскольку он почти никаких ограничений не накладывает, но тем не менее, можно говорить об его достоинствах и недостатках. Достоинства — прежде всего, близость к предметной области и «обычному» человеческому мышлению. Сущности предметной области и так имеют своё состояние и поведение, это более-менее прямолинейно ложится в код. Кроме того, с точки зрения техники программирования каждый объект независим, так что его можно разрабатывать отдельно, и главное, думать о нём отдельно: рассматривать систему как набор взаимодействующих агентов, которые вместе решают задачу, но каждый из них более-менее обособлен, его можно отдельно разрабатывать и отдельно тестировать. Ещё одно важное тактическое соображение — реализацию объектов можно смело менять, пока выполняются их инварианты, во внешнем мире это гарантированно ничего не сломает.

Есть и недостатки. Главный — это, пожалуй, отсутствие строгих топологических ограничений на связи между объектами, что приводит к тенденции «чистых» объектно-ориентированных программ превращаться в месиво из объектов, где каждый вынужден

знат о каждом. Поэтому при программировании в чистом объектно-ориентированном стиле обязательно создание высокоуровневой структуры системы или применение дополнительных стилей, которые её диктуют, например, слоистого.

Второй недостаток (по сравнению с функциональным программированием, по крайней мере, и даже со структурным) — склонность к побочным эффектам. Методы обычно меняют состояние объекта, так что разная последовательность вызовов одних и тех же методов с одними и теми же параметрами приводит к разным результатам. Это существенно усложняет анализ.

3.2. Слоистый стиль

Перейдём к рассмотрению «настоящих» архитектурных стилей. Первый такой стиль, а точнее даже группа стилей — это слоистый стиль и его возможные вариации. Суть этого стиля в том, что мы разделяем систему на слои, где каждый слой может пользоваться слоями ниже и предоставляет интерфейс для слоёв выше, при этом сам ничего о них не зная. Его можно понимать как «многоуровневый клиент-сервер» в том смысле, что каждый слой выступает клиентом слоёв ниже и сервером для слоёв выше. Компонентами в таком стиле выступают сами слои, они могут быть сколь угодно сложно устроены внутри, но это их детали реализации. Соединителями — протоколы общения слоёв (или просто программные интерфейсы). Примеры слоистых архитектуры мы уже видели: трёхзвенная архитектура, также по такому принципу устроены сетевые стеки (модели OSI и TCP/IP состоят из слоёв протоколов), операционные системы, многие бизнес-приложения.

Преимущества слоистого стиля — это в первую очередь постепенное повышение уровня абстракции от низких уровней к высоким. В строгом варианте, когда слой может общаться только со слоем непосредственно ниже, это позволяет вообще не думать о реализации всей системы, а просто программировать в терминах предоставляемой слоем абстракции (так, например, реализуются сетевые приложения, вы просто пользуетесь сокетами или высокоуровневыми протоколами и не интересуетесь тем, что происходит на уровнях глубже).

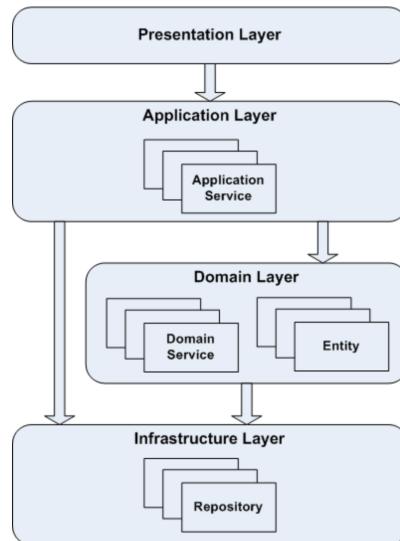
Ещё слоистость существенно облегчает сопровождение системы. Поскольку каждый уровень влияет только на уровни выше, влияние каждого изменения легко оценить и отследить. При этом можно использовать разные реализации каждого уровня, лишь бы они удовлетворяли общему интерфейсу — примером этого снова являются сетевые приложения, мы можем использовать на физическом уровне Ethernet или WiFi, совершенно ничего не меняя уровнями выше. То же касается драйверов операционной системы — мы можем использовать низкоуровневую графическую библиотеку типа OpenGL для вывода графики, не зная, что видеокарта у пользователя, был бы драйвер. Да и саму библиотеку можно поменять (на DirectX или Vulcan), если у нас есть высокоуровневый графический движок, обрабатывающий низкоуровневые интерфейсы (например, Unity или UnrealEngine).

Однако есть и проблемы. Во-первых, уровневый стиль оказывается не всегда применим — взаимодействие между элементами системы может быть таким, что не позволяет себя упорядочить по уровням. Пример тому был на первой лекции, с ПО к осциллографу, там уровневый стиль формально можно было навести, но данные шли «вверх» по уровням, а команды от пользователя — «вниз», так что все части системы всё равно были вынуждены знать про все остальные. Такие ситуации в реальной жизни встречаются, и тогда пытаться натянуть уровневый стиль на систему не стоит.

Во-вторых, проблемой может стать производительность системы. Сложные уровневые архитектуры имеют тенденцию обрастиать функциями, которые просто прокидывают запрос на уровень ниже, что ведёт к ненужному оверхеду на вызовы. Если производительность критична, уровневый стиль может всё-таки хорошо подойти, но может и нет (особенно, если перестараться с количеством уровней).

Тем не менее, уровневость — это хорошо, поэтому иногда стоит даже проделать дополнительную работу, чтобы разделить систему на уровни. Уровневая архитектура считается довольно стандартной и, например, профстандарт профессии «архитектор» даже подразумевает её как архитектуру по умолчанию.

Классический пример слоистой организации системы применяется в методологии проектирования «Domain-Driven Design», о которой подробно речь пойдёт чуть дальше в курсе. Основа методологии — выделение отдельного уровня предметной области, где размещены все бизнес-объекты, то есть код, который собственно делает полезную работу в предметной области, без привязки к конкретному UI или техническим вещам типа сохранения в базу данных. Типичное приложение, сделанное по DDD, помимо уровня предметной области имеет ещё уровни представления, приложения и инфраструктурный:



© http://uniknow.github.io/AgileDev/site/0.1.8-SNAPSHOT/parent/ddd/core/layered_architecture.html

- Уровень *представления* отвечает за UI, не хранит состояние и не содержит никакой содержательной логики, кроме, быть может, анимации элементов управления и т.п.
- Уровень *приложения* отвечает за пользовательские сценарии и оркестрацию (страшное слово, на самом деле просто координацию) объектов уровня предметной области.
- Уровень *предметной области* отвечает за модель предметной области и только за неё — там находятся объекты, моделирующие сущности реального мира и логику работы системы. Уровень предметной области может переиспользоваться между различными приложениями.

- *Инфраструктурный уровень* отвечает за вещи, которые не специфичны даже для целой предметной области — хранение данных, сетевое взаимодействие, разные утилиты типа работы с валютами. Там же находятся сторонние библиотеки (в терминах которых вполне могут быть реализованы уровни выше, то есть форма на уровне представления вполне может наследоваться от библиотечного класса инфраструктурного уровня — наследование идёт снизу вверх).

Внутри уровня предметной области также могут находиться уровни, на которые разделена модель. Например, уровень принятия решений, операционный и ресурсный — чтобы разделить классы, управляющие бизнес-процессом, следящие за его текущим состоянием и хранящие данные об оборудовании и остатках материалов, например.

Надеюсь, что общая идея понятна, дальше в курсе мы подробно поговорим про DDD, и к его уровневой модели тоже вернёмся.

3.2.1. Клиент-сервер

«Клиент-сервер» — это в каком-то смысле вырожденный случай уровневой архитектуры, когда уровня всего два. Собственно, клиенты и сервер — это компоненты такого архитектурного стиля, сетевые протоколы (обычно) — соединители. Ограничения — клиенты не могут общаться друг с другом и могут общаться только с сервером, сервер ничего не знает о клиентах до того момента, как они не начнут с ним взаимодействовать, даже их количество.

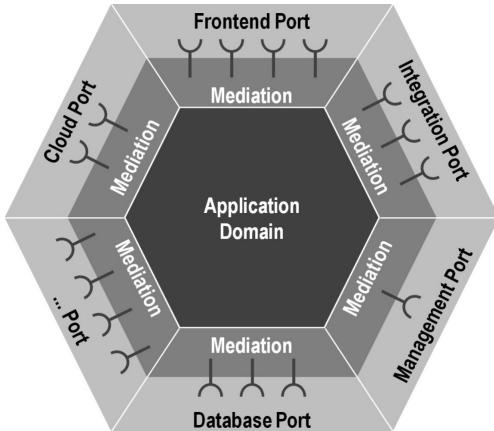
Серверов, кстати, может быть много, это позитивно оказывается на масштабируемости системы, но привносит дополнительные трудности синхронизации состояния серверов, о которых будет чуть подробнее позже, в части курса, относящейся к распределённым приложениям.

Стиль типичен для несложных веб-приложений, где клиентом обычно выступает браузерная часть приложения, мобильных сетевых приложений или, что может быть несколько неожиданно, операционных систем. Графическая подсистема Linux, например, реализована по клиент-серверной архитектуре, есть оконный сервер и приложения, которые шлют ему запросы на отрисовку графических примитивов.

3.2.2. Гексагональная архитектура

Гексагональная архитектура, или «порты и адаптеры» — это дальнейшее развитие уровневой архитектуры. В классической уровневой архитектуре уровень предметной области, где находится содержательная бизнес-логика системы, находится где-то посередине — над ним слои взаимодействия с пользователем, под ним — инфраструктура, библиотеки, персистентность² и т.п. В гексагональной архитектуре предлагается инфраструктуру тоже рассматривать как часть внешнего мира, и тогда система получается устроенной очень просто. Самый нижний уровень — уровень предметной области. Над ним — слои, обеспечивающие взаимодействие с внешним миром, которые включают в себя предоставляемые и потребляемые интерфейсы, и адаптеры, преобразующие данные из внешнего мира в чистую предметно-ориентированную модель:

² Т.е. по сути сохранение в БД.



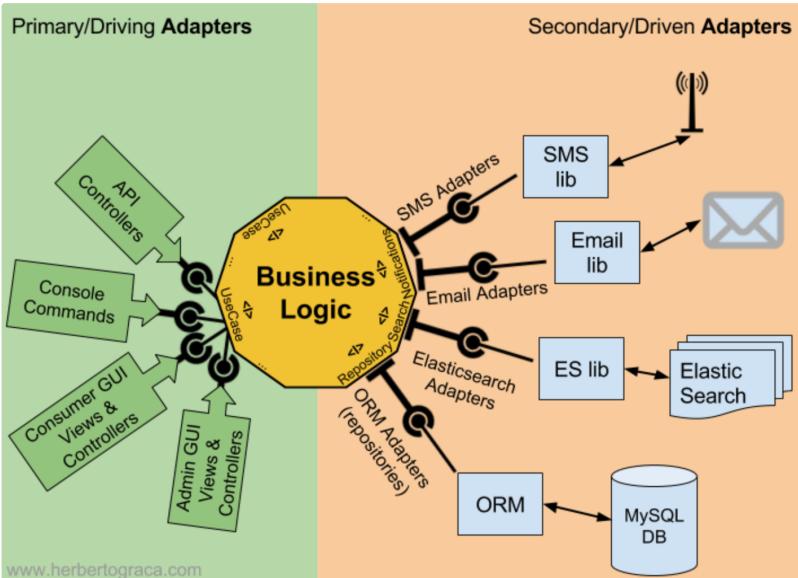
© B Butzin et al, Microservices Approach for the Internet of Things

Основная идея подхода состоит в том, что классы, моделирующие предметную область, были максимально просты и максимально переиспользуемы. То есть если мы хотим сделать веб-приложение, десктопный и мобильны варианты — не вопрос, классы предметной области вообще не надо менять. Если хотим сменить СУБД с реляционной на объектно-ориентированную — не вопрос, классы предметной области менять не надо.

Чтобы этого добиться, слой предметной области описывает интерфейсы, которые он предоставляет, и которые ему нужны для работы. Слой адаптеров реализует эти интерфейсы, что само по себе может быть сложно, но предметной области не касается — например, Data Access Layer может полностью инкапсулировать в себе работу с БД. Это на самом деле пример применения принципа Dependency Inversion — и для того, чтобы собрать всё воедино, используется подход Dependency Injection. То есть классам предметной области в конструкторы передаются конкретные реализации потребляемых ими интерфейсов, а внешнему миру передаются конкретные классы предметной области или адаптеры, реализующие интерфейсы, которые компонент обязуется реализовывать. Почему эта архитектура и известна как «порты и адаптеры» — всё взаимодействие с классами предметной области осуществляется через порты, вся валидация и преобразование данных — через адаптеры, и никак больше.

Вот почему архитектура называется гексагональной, вопрос менее понятный. На самом деле, один компонент может предоставлять много разных наборов интерфейсов, не обязательно шесть. Но так уж повелось, что компонент в работах по этому делу рисовали в виде шестиугольника. К тому же, шестиугольниками можно замостить плоскость, что иллюстрирует взгляд на систему в целом как на кучу таких вот компонентов-шестиугольников, интегрированных (через порты, естественно) друг с другом. Поэтому название прижилось.

Вот немного более страшная картинка, которая показывает пример типичного бэкенда веб-приложения:



© <https://herbertograca.com/2017/09/14/ports-adapters-architecture/>

Тут показана ещё довольно важная штука — порты у компонента делятся на предstawляемые (они же «primary» или «driving») и потребляемые (они же «secondary» или «driven»). Инициация действия возможна только через primary-порты, тогда как secondary-порты используются для реагирования на запросы и ими управляет ядро системы.

На рисунке видно, что в качестве primary-портов могут выступать интерфейсы, используемые «средствами доставки» запросов — сетевыми запросами разных видов, консольными командами. Однако сами порты ничего не знают про средства доставки, и им дела до них нет. Различные порты соответствуют различным случаям использования приложения — например, как на рисунке, пользовательской функциональности, функциональности админа, функциональности API.

В качестве secondary-портов выступает вся служебная функциональность, нужная системе. Причём, как обычно, система ничего не знает про то, как эта функциональность реализуется — в ней есть лишь функциональность нотификаций, поиска и хранения, например. Конкретные библиотеки и технологии спрятаны за адаптерами, которые, в свою очередь, реализуют интерфейсы из ядра. Например, интерфейс нотификации реализуется адаптером электронной почты, который дёргает библиотеку, отправляющую письма. Ядро даже не знает, куда именно отправляет нотификации, потому что другая реализация этого же интерфейса может слать SMS-ки или сообщения в Slack.

У гексагональной архитектуры есть очевидные плюсы.

- Изоляция содержательного кода системы от «механизмов доставки» — от того, каким образом происходит взаимодействие с пользователем. Что позволяет легко заменить фронтенд или даже переиспользовать одну модель предметной области из разных приложений. Что, впрочем, типично не только для гексагональной архитектуры, но и для любой уровневой архитектуры вообще.

- А вот что классическая уровневая архитектура не умеет — это изоляция вспомогательных механизмов. Это делает модель предметной области независимой не только от конкретных библиотек и инструментов, но даже от прослоек типа Data Access Layer. Что хорошо не только в плане «поменять технологию на совсем другую», но, главное, позволяет держать модель предметной области максимально простой.
- Есть и чисто прагматическое преимущество такого разделения — лёгкость тестирования. Если ядро системы напрямую ни от кого не зависит, во все его внешние зависимости можно передать объекты-заглушки³ и писать модульные и интеграционные тесты, не разворачивая реальную СУБД, не устанавливая специфическое оборудование и т.д. и т.п.
- Раз всё взаимодействие с внешним миром происходит через адаптеры, адаптеры могут заниматься верификацией, валидацией и конвертированием данных во внутреннее представление системы. Так что модель предметной области может спокойно хранить данные в том формате, в котором ей удобно, и считать, что данные, которые туда таки попали, заведомо корректны.

Есть, конечно, и минусы:

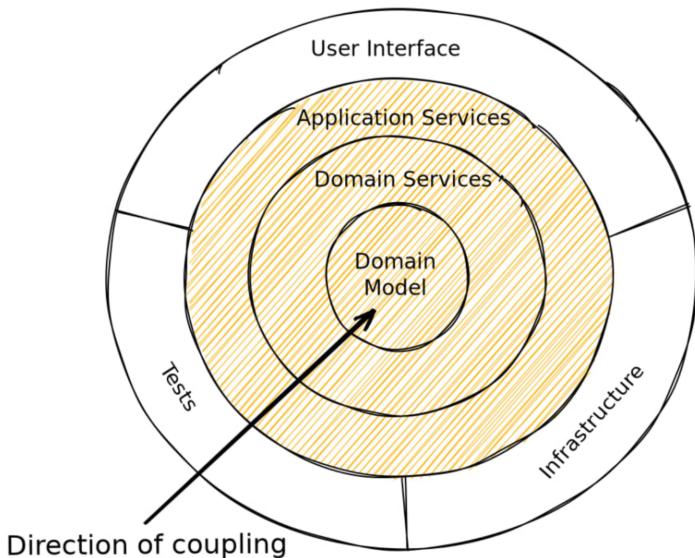
- Иногда гексагональная архитектура — это стрельба из пушки по воробьям. Если приложение планируется всего одно, предметная область проста, а менять технологии не планируется (хотя это всегда не планируется...), то можно потратить кучу времени на написание красивой модели, адаптеров, на внедрение зависимостей и т.д. и т.п. Часто оно того не стоит.
- Некоторая тонкость заключается в том, что считать платформой, на которой пишется ядро системы, а что внешней зависимостью, которую надо внедрять. Считать зависимостью стандартную библиотеку, наверное, довольно тупо. Но вот Guava в Java или boost в C++ — это зависимость или часть платформы? Или вот библиотеки типа Qt — на них, в общем-то, пишется приложение от начала до конца, и там куча инфраструктурных вещей реализована — их надо внедрять? Если да, то как? Но если нет, то где грани между внешним миром и платформой? И много ли сейчас приложений пишется на голом языке программирования, без какого-то фреймворка и технологического стека вокруг него, со своими библиотеками, инструментами и даже иногда языками?
- Гексагональная архитектура не то чтобы очень специфична. Она говорит, что есть ядро, есть адаптеры, есть внешний мир, и работает это всё вместе так-то. Это хорошо, но при её применении остаётся слишком много свободы, что скорее плохо, чем хорошо.

3.2.3. Луковая архитектура

Луковая архитектура («Onion architecture») — архитектура с довольно странным названием, являющаяся, на самом деле, уточнением гексагональной архитектуры. Идея та-

³ Моск-объекты, если более точно.

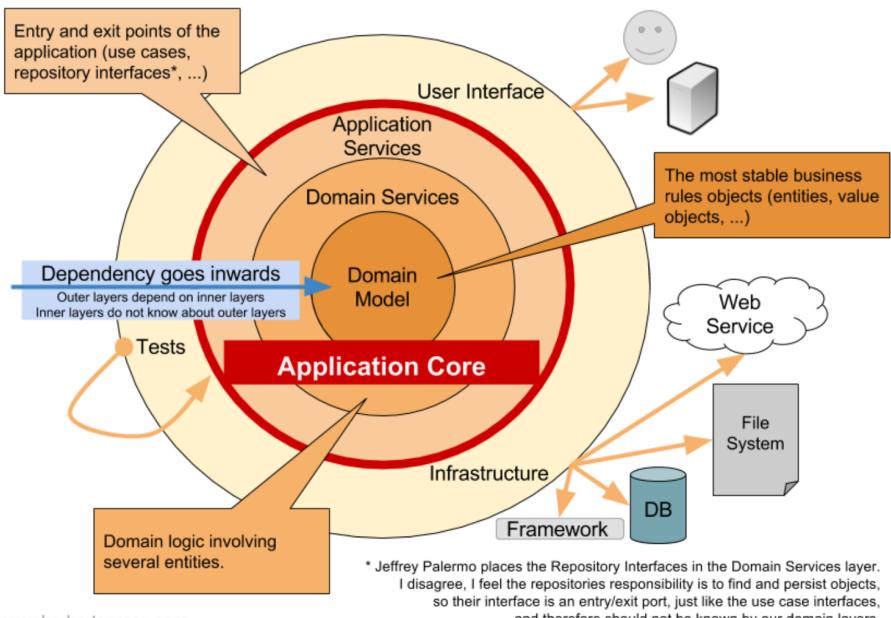
кая же — есть модель предметной области, включающая в себя всю бизнес-логику и модель данных приложения (не только приложения, а целой системы — ведь ядро может переиспользоваться в нескольких приложениях сразу). Есть внешний мир, который с ядром взаимодействует. Однако она предписывает наличие в ядре дополнительных уровней:



© <https://dev.to/barrymcauley/onion-architecture-3fgl>

Общее правило структурирования уровней тут понятное: внутренние слои ничего не знают о внешних, а лишь предоставляют интерфейсы и принимают в конструкторы зависимости по этим интерфейсам. В самом центре находится доменная модель (та самая модель предметной области) — классы, реализующие бизнес-логику и моделирующие сущности из реального мира. Над ней — доменные сервисы, то есть классы, использующие несколько сущностей в своей работе (функциональность, которую нельзя естественным образом отнести к одному из классов предметной области). Часто сервисы — это статические классы. Над слоем доменных сервисов — сервисы приложений, они занимаются оркестрацией объектов и сервисов предметной области, реализуют случаи использования, всячески поддерживают код, который пользуется компонентом из внешнего мира. Над слоем сервисов приложений находится уже внешний по отношению к компоненту мир — пользовательский интерфейс и механизмы доставки, инфраструктура, хранение данных и т.д. Уровненность нестрогая, то есть уровень вправе обращаться к объектам любого из уровней ниже.

Кстати, структура уровней в «луковой архитектуре» примерно соответствует уровням, предлагаемым в Domain-Driven Design. Единственное, что DDD разделяет инфраструктурный уровень и уровень приложения, тогда как тут они смешаны в один, поскольку и то и другое — штуки для поддержки внешних зависимостей компонента. Зато в DDD, как мы позже узнаем, доменные сервисы являются полноценными обитателями доменной модели, а тут они вынесены в отдельный слой. Вот несколько более подробный рисунок:



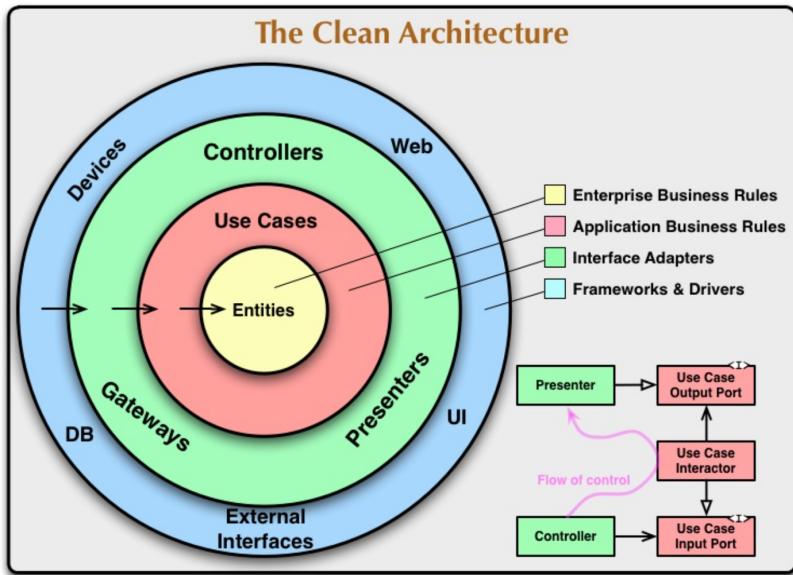
www.herbertograca.com

© <https://herbertograca.com/2017/09/21/onion-architecture/>

Тут автор (очень рекомендую его цикл постов с подробным рассказом о том, откуда всё это пошло и как эволюционировало) обращает внимание, что в оригинале в луковой архитектуре репозитории были частью доменной модели, но на самом деле логично, чтобы они были вовне, на самом внешнем уровне (как в гексагональной архитектуре). Поэтому про луковую архитектуру теперь, в общем-то, обычно так и пишут.

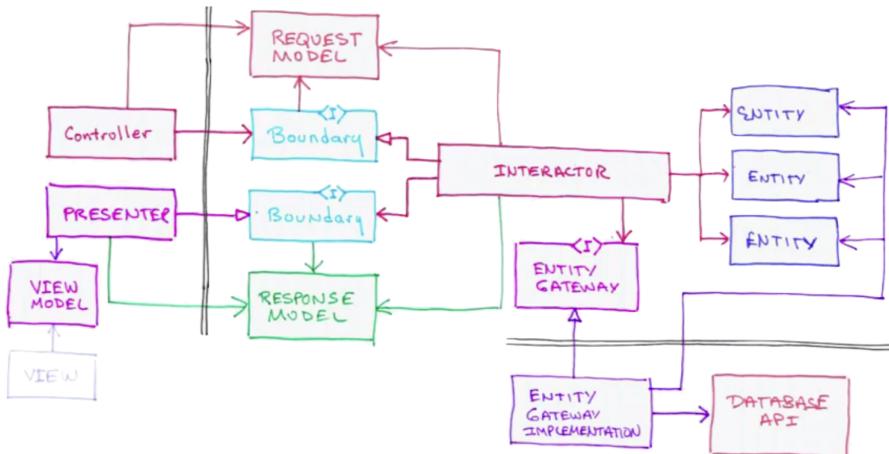
3.2.4. Чистая архитектура

«Чистая архитектура» («Clean architecture») — дальнейшее уточнение идей луковой и гексагональной архитектур. Тоже предполагается слоистый стиль, где самый нижний слой — доменная модель, тоже зависимости направлены строго от внешних слоёв к внутренним, тоже используется Dependency Injection, чтобы классы доменной модели могли работать с внешним миром. Но в дополнение ко всему этому она специфицирует поток управления и определяет конкретные способы взаимодействия с пользователем:



© <https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/>

На самом деле, самое интересное тут — как раз взаимодействие с внешним миром, детали статического устройства которого показаны на этом рисунке:



© <https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/>

Двойные чёрные полоски — это границы компонента, отделяющие плохой внешний мир от хорошего внутреннего. Что происходит в рантайме:

1. Запрос попадает в контроллер (не важно откуда, приходит по сети или из GUI десктопного/мобильного приложения, всё равно есть какая-то точка входа, которая обслуживает команды от пользователя).

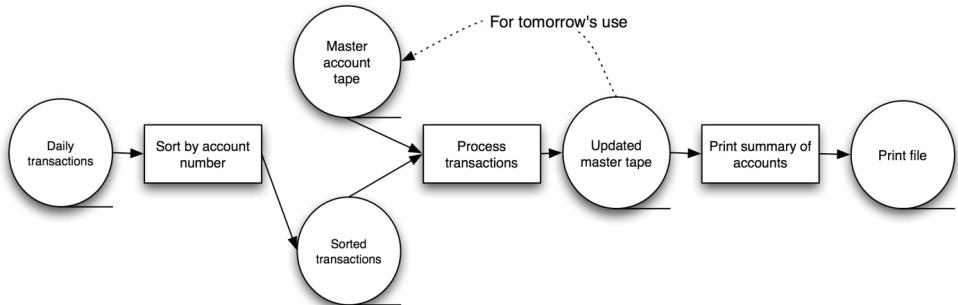
2. Контроллер парсит параметры запроса (архитектура больше всё-таки про веб-приложения), либо выполняет валидацию, и конвертирует запрос в Request Model, которая уже провалидирована и в виде, удобном для ядра системы.
3. Дальше контроллер дёргает интерактор, передавая ему Request Model в качестве параметра вызова одного из его методов. Контроллеру интерактор передаётся при инициализации системы по его интерфейсу Boundary (Dependency Injection тот самый).
4. Интерактор с помощью репозитория Entity Gateway (про который он опять-таки знает только его интерфейс, и реальный объект подставляется через DI) находит нужные для исполнения запроса сущности.
5. Репозиторий может подгрузить сущности из базы, создать при необходимости или просто вернуть уже готовые, это его дело, и это находится вне границ системы (хотя сами сущности — в самом её центре, такие дела).
6. Интерактор координирует работу сущностей по исполнению запроса, после чего формирует Response Model.
7. Response Model передаётся презентеру, про который интерактор опять-таки знает только его интерфейс.
8. Презентер, используя данные из Response Model, генерирует View Model и отправляет её клиенту (в случае, если речь идёт про настольное/мобильное приложение, там тоже бывают View Model-ы, на которые байдытся элементы на форме, либо презентер может сам управлять отображением информации на форме — зависит от используемой библиотеки и соответствующих ей паттернов).
9. У клиента View отображает обновившиеся данные.

В общем-то, ничего особо нового в этой архитектуре нет, но чёткое разделение ответственности сильно помогает и в разработке, и в тестировании. Поэтому такая архитектура стала весьма популярной. Например, есть паттерн VIPER — View, Interactor, Presenter, Entity, Routing — это буквально реализация идей «Чистой архитектуры» с некоторыми небольшими модификациями для программирования iOS-приложений. Там в основном только так и пишут.

3.3. Стили, ориентированные на поток данных

3.3.1. Пакетная обработка

Пакетная обработка — «прадедушка архитектурных стилей», применявшийся ещё на заре массовой информатизации в финансовых системах глубокой древности:



© N. Medvidovic

Данные в те времена хранились на магнитных лентах (а это медленные устройства с последовательным доступом, перемотать ленту было целым делом). Данные о транзакциях за день подавались на вход программе сортировки, которая сортировала их по номерам аккаунтов, после чего выгружались на ленту с отсортированными транзакциями. Затем они и лента с данными об аккаунтах подавались на вход программе, которая проводит транзакции и сливает их в обновлённую ленту с аккаунтами (как в сортировке слиянием, для этого и надо было сначала транзакции отсортировать). Затем то, что получилось, подавалось на вход программе-печаталке, которая выдавала статистику по аккаунтам.

Итого, система в стиле «пакетная обработка» строится как набор отдельных программ, которые выполняются последовательно, обмениваясь данными средствами операционной системы. Это давно уже не ленты, а файлы, либо pipes или named pipes⁴. При такой схеме между процессами требуется передавать в явном виде всё, что необходимо им для работы — сами данные, конфигурацию, управляющие команды.

Стиль хоть и древний, но очень популярен до сих пор. По сути, Linux way с большим количеством маленьких утилит, из которых с помощью пайпов выстраиваются конвейеры — это и есть пакетная обработка. Так что этот стиль применяется практически во всех скриптах под Linux и без него трудно представить работу системных администраторов, DevOps и т.д.

Важное преимущество этого стиля — независимость отдельных программ. Они могут быть написаны на каком угодно языке и какой угодно технологии, лишь бы умели читать из входного потока. Можно их вообще не писать, а переиспользовать уже готовые в любой части конвейера. Этакая микросервисная архитектура внутри одной машины.

Важный недостаток этого стиля — неторопливость работы. Каждая программа — отдельный процесс, даже просто их запуск трудоёмок по времени и памяти, да и коммуникации между процессами, хоть и не так тяжелы, как коммуникации по сети, всё-таки гораздо медленнее общения потоков внутри процесса.

⁴ Абстракция чего-то среднего между файлом и очередью сообщений, в неё можно писать одним процессом и вычитывать данные другим. Реально данные на диск не сохраняются, поэтому пайпы гораздо быстрее файлов как средство общения между процессами. Кстати, пайпы (даже именованные) поддерживаются и Windows, просто в Linux ими пользоваться удобнее

3.3.2. Каналы и фильтры

Каналы и фильтры (или «pipes and filters») — стиль, в котором программа представляется в виде набора фильтров, которые как-то преобразуют данные, идущие по каналам. При этом фильтры независимы друг от друга, то есть не имеют разделяемого состояния и ничего не знают про фильтры до и после них. Всё, что они видят — это данные в своих входных каналах. Собственно, фильтры являются единственным типом элементов в такой архитектуре, а каналы — единственным типом соединителей.

«Каналы и фильтры» похожи на «пакетную обработку», но не требуют, чтобы каждый фильтр был отдельной программой, что помогает победить проблемы с производительностью в пакетной обработке. Кроме того, каналы и фильтры имеют тенденцию образовывать сложные сети, в отличие от пакетной обработки, где всё в основном линейно.

Бывают варианты каналов и фильтров:

- конвейеры — где фильтры связаны просто в линейную цепочку, очень топологически простой стиль, подходящий для несложной логики обработки (хотя сами фильтры могут быть сколь угодно сложны);
- ограниченные каналы — где канал представляет собой очередь с ограниченным количеством элементов, блокирующую фильтр-источник, если очередь переполнена. На самом деле, лучше ограниченность каналов иметь в виду всегда, потому что фильтры, обрабатывающие данные с разной скоростью, могут привести к «пробкам» из данных на разных этапах обработки.
- Типизированные каналы — где каналы знают тип передаваемых данных, и фильтры могут подключаться только к каналам правильного типа. Именно такой стиль в итоге был выбран в первой лекции этого курса в примере про осциллограф.

Преимущества этого стиля таковы.

- Поведение системы — это просто последовательное применение поведений компонентов. Так что о нём легко рассуждать, его легко понять, такие системы легко поддерживать.
- Легко добавлять, заменять и переиспользовать фильтры. Если не принимать в расчёт типизированные каналы, то вообще любые два фильтра можно использовать вместе. Если принимать, то любые два фильтра, у которых подходящие типы «портов», можно использовать. Специально продумывать интеграцию компонентов не нужно, она получается сама собой.
- Широкие возможности для анализа. Поскольку есть чёткие ограничения на потоки данных, систему можно рассматривать просто как граф из фильтров с рёбрами-каналами, что делает применимыми все алгоритмы анализа графов. Можно считать пропускную способность системы, задержки (среднюю и максимальную), искать взаимные блокировки в сложных сетях.
- Широкие возможности для параллелизма. Каждый фильтр может работать одновременно со всеми остальными, либо в отдельном потоке, либо в отдельном процессе на другой машине (что, кстати, делает фильтры естественными кандидатами в микросервисы).

Недостатки тоже есть:

- Последовательное исполнение — что странно противоречит достоинству про параллелизм. Но пока первые фильтры из сети не сделают своё дело, следующие за ними к работе приступить не могут. Это не важно, когда данных много и вся сеть занята их обработкой, но если данные поступают лишь иногда, они должны последовательно пройти через все фильтры, при этом большая часть фильтров будет простаивать.
- Проблемы с интерактивными приложениями, поскольку данные идут по фильтрам в одном направлении и непонятно, как ими управлять. Можно придумать «обратные» каналы управления, как это было в примере из первой лекции, но об этом надо специально думать и это несколько портит стройную картину этого стиля.
- Пропускная способность всей системы определяется самым “узким” элементом. Опять-таки, это можно обойти, масштабировав медленный фильтр, но это может быть технически непросто и об этом надо вовремя подумать.

3.4. Blackboard

Blackboard — это архитектурный стиль с общей памятью. Есть центральное хранилище данных, тот самый «Blackboard», есть компоненты, которые знают только про Blackboard, не имеют своего состояния и, собственно, выполняют полезную работу. Процесс работы системы устроен так, что каждый компонент смотрит на Blackboard, (возможно) находит там данные, которые может преобразовать, преобразовывает их и записывает обратно на Blackboard, в надежде, что какой-то другой компонент сможет преобразовать их дальше. Это чем-то похоже на группу экспертов, которые решают задачу, сидя в одной комнате с доской — каждый эксперт шарит только в своей предметной области, поэтому выходит к доске и пишет на ней что-то, что соответствует его экспертизе и, возможно, поможет другим экспертам решить задачу.

Весь процесс вычислений управляет только через Blackboard, поэтому если мы хотим, чтобы некоторые действия выполнялись последовательно, нам надо хранить на Blackboard какой-то токен исполнения и обновлять его при выполнении действий.

Такой подход хорош тем, что компоненты могут быть полностью независимы и работать параллельно (и разрабатываться независимо, кстати), система легко масштабируется и расширяется добавлением новых компонентов, и главное, что вам не надо даже понимать алгоритм решения задачи — вы просто бросаете в систему компоненты пока задача не решится. Это же и главный недостаток этого стиля — задача легко может и не решиться. Поэтому такой стиль применяется прежде всего в системах искусственного интеллекта, где задача и так вполне может быть нерешаемой. Ещё, кстати, Blackboard, как правило, является узким местом системы, поскольку ему надо синхронизировать доступ независимым компонентам к себе.

Кстати, родственен Blackboard стиль, основанный на правилах переписывания — например, машины Маркова и язык Рефал, графовые грамматики и подобные штуки. Там тоже есть центральная структура данных и набор правил, которые её независимо и постепенно изменяют. Такой подход вполне распространён и в «обычной» программной инженерии.

3.5. Событийный стиль

Событийные стили (или *стили с неявным вызовом*) — общее название разных вариантов стилей, где используются оповещения вместо явных вызовов методов. Во всех таких стилях есть «слушатели», которые могут подписываться на события, и при наступлении события система сама вызывает всех зарегистрированных слушателей.

В таких архитектурах компоненты имеют два вида интерфейсов — обычный набор методов, и события, на которые можно подписываться. В качестве соединителей используются либо прямые вызовы методов, либо неявные вызовы слушателей по наступлению события (почему стили и называются стилями с неявным вызовом).

Инварианты всех таких стилей:

- те, кто производит события, не знают, кто и как на них отреагирует — они, как правило, технически имеют список подписавшихся, но обычно не вправе даже узнать их количество, не говоря уж о том, чтобы что-то делать с подписавшимися напрямую;
- не делается никаких предположений о том, как событие будет обработано и будет ли вообще — источник просто нотифицирует систему о наступлении события, а уж подписан на него кто-нибудь или нет, в каком порядке кто подписан и т.д. — не его дело.

Преимущества всех таких стилей — это переиспользуемость компонентов и лёгкость конфигурирования системы. Высокая переиспользуемость достигается за счёт очень низкой связности между компонентами, ведь источник событий вправе вообще ничего не знать о тех, кто им пользуется. Лёгкость конфигурирования, как во время компиляции, так и во время выполнения, достигается за счёт того, что подписки на события можно легко менять, меняя при этом всю функциональность системы.

Недостатков, тем не менее, тоже довольно много.

- Зачастую неинтуитивная структура системы. Без применения дополнительных архитектурных ограничений подписки на события превращаются в хаотичный клубок, в котором хаотично распространяются нотификации. Поэтому мы и рассмотрим конкретные стили, накладывающие дополнительные ограничения.
- Компоненты не управляют последовательностью вычислений. Работа системы состоит в генерации событий и реакций на события, и делать что-то в правильном порядке в сколько-нибудь сложной системе может оказаться проблематичным.
- Непонятно, кто отреагирует на запрос и в каком порядке придут ответы. Компонент, генерирующий события, не вправе предполагать, что на событие кто-то отреагирует, поэтому если это событие, например, «мне нужны данные для дальнейшей работы», мы не вправе рассчитывать на ответ. А если надо запросить несколько разных источников, то неизвестно, кто и когда ответит. Поэтому такие системы принципиально асинхронны.
- Тяжело отлаживаться. Вы не можете просто сделать `step into` при вызове метода, вы должны мучительно ковыряться в списке подписчиков. Некоторые среды, типа C#/Visual Studio, хорошо поддерживают отладку событий, но даже там, когда управление прыгает по всему коду, отлаживаться тяжело. К тому же, событийные системы принципиально асинхронны, что создаёт дополнительную боль.

- Ситуации, очень похожие на гонки, даже если у вас всего один поток. Классическая гонка — это когда результат работы программы зависит от случайного порядка переключения потоков планировщиком. Гонка в событийных системах — это когда результат работы программы зависит от случайного порядка вызова обработчиков при нотификации. Обычно событийные системы хоть и не позволяют закладыватьсь на определённый порядок вызова обработчиков, всё же вызывают их в порядке подписывания, что делает процесс хоть сколько-нибудь детерминированным. Но, во-первых, это не всегда возможно (например, в распределённых системах — кто первый получил событие, тот его и обработал), во-вторых, порядок подписывания тоже такой себе ориентир — на событие могут подписываться разные компоненты в разных частях кода, и помнить, кто когда должен подписаться, может оказаться слишком хлопотно. В таких местах особо часто появляется и особо опасен анти-паттерн «Sequential coupling».

3.5.1. Издатель-подписчик

«Издатель-подписчик» (или «Publish-subscribe») — самый простой подвид стилей с неявным вызовом, и вместе с тем весьма популярный и эффективный. Есть издатели, они публикуют сообщения (синхронно или асинхронно, то есть дожидаясь их обработки либо нет). Есть подписчики, которые подписываются на издателей и получают от них события. Есть маршрутизаторы, задача которых — быть посредниками между издателями и подписчиками, фильтровать и маршрутизировать сообщения. При этом в самых простых конфигурациях без маршрутизаторов прекрасно обходятся, но они могут быть полезны: например, маршрутизатор может по очереди отправлять сообщения то одному подписчику, то другому, реализуя тем самым балансировку нагрузки.

Компонентами в таком архитектурном стиле являются издатели, подписчики и маршрутизаторы, соединителями — сетевые протоколы или *очереди сообщений*, либо, если дело происходит на одной машине, механизмы наподобие паттерна «Наблюдатель». В качестве данных в этом стиле выступают подписки, нотификации о произошедших событиях, публикуемая издателями информация. Ограничения — издатели ничего не знают о подписчиках, подписчики, как правило, ничего не знают друг о друге, только об издателях.

Преимущества этого стиля, как и обычно для событийно-ориентированных схем — очень низкая связность между компонентами, но при этом высокая эффективность распространения информации, за счёт свойственной этому стилю простоты топологии.

3.5.2. Событийно-ориентированный стиль с шиной

Событийно-ориентированный стиль с шиной предполагает наличие шин событий, через которые могут общаться компоненты. При этом всё остальное как в обычном событийно-ориентированном стиле: есть компоненты, которые могут генерировать какие-то события и подписываться на чужие, но разница в том, что компоненты принципиально не могут взаимодействовать друг с другом напрямую, а общаются только через шину. Соответственно, подписаться можно только на события вшине, и посыпать события можно только шине. При этом возможны два варианта взаимодействия — «push», когда шина сама активно уведомляет своих подписчиков, и «pull», когда компоненты время от времени опрашивают шину на предмет наличия в ней новых событий. Шина в системе вполне

может быть одна, но часто это несколько именованных шин с разными типами данных.

Преимущества такого подхода — это ещё большая независимость компонентов, лёгкость масштабирования и добавления новой функциональности. Если кто-то не успевает обрабатывать запросы — не проблема, подключим к шине ещё одну копию. Если нам надо новую функциональность, мы просто пишем компонент, подключаем его к шине и ничего больше в системе обычно менять не надо. Особенно эффективен такой стиль для распределённых приложений, где каждый компонент может быть отдельным сервисом.

Ещё событийные шины позволяют применять приём, называющийся «Event Sourcing» — когда состояние системы не хранится в явном виде, а хранится append-only лог его изменений, по которому можно однозначно восстановить текущее состояние. Это делает более явным управление состоянием в распределённых системах — явное состояние корректно синхронизировать трудно (есть теорема CAP⁵, которая говорит, что можно выбрать только два из трёх: консистентность данных, их доступность или устойчивость к разделению), события по шине разослать легко. Разные компоненты, конечно, могут получить событие в разное время, но для системы будет выполняться свойство «eventual consistency» — данные будут согласованы в какой-то момент в будущем. Причём, мы даром получаем лог, по которому видно, как система пришла в такое состояние и кто виноват, если что-то пошло не так, и получаем, даром же, возможность откатиться до любого предыдущего состояния. Платить за это приходится скоростью работы, но это отчасти компенсируется созданием сnapshotов — полных состояний, которые каждый компонент может сам делать по логу, и использовать их как базу для быстрого построения следующих состояний. В конце концов, системы контроля версий именно так и работают, и ничего.

Пример дальнейшего уточнения такого стиля — архитектурный паттерн интеграции «Enterprise Service Bus», умная шина, позволяющая преобразовывать данные в универсальное внутреннее представление и выгружать их в виде, пригодном для каждого компонента. Там обычно компонентами выступают третьясторонние приложения, типа электронных таблиц, бухгалтерских систем и т.п. — мы даже не можем менять код компонентов, но с помощью шины можем добиться их эффективной интеграции.

3.6. Peer-to-peer

Стиль «Peer-to-peer», или одноранговая сеть, — стиль, при котором система состоит из большого количества приложений, каждое из которых в принципе может само решать все задачи и является самодостаточным, но чем больше приложений работают совместно, тем шире возможности системы. Компоненты в таком стиле — это отдельные приложения, которые имеют своё состояние, свой поток управления и всё, что нужно для работы. В качестве соединителей, как правило, выступают сетевые протоколы, а в качестве элементов данных — сообщения по сети.

Топологических ограничений на связи между компонентами не накладывается, даже наоборот, приветствуются избыточные связи. Кроме того, предполагается, что топология сети может динамически изменяться во время работы. Компоненты могут выходить из сети или появляться новые.

Основное преимущество Peer-to-peer — это отказоустойчивость. Можно хоть физически уничтожить часть системы, остальная часть продолжит работать, хоть и менее эффективно.

⁵ Мы поговорим про всё это подробнее ближе к концу курса, когда речь пойдёт про архитектуру распределённых приложений.

тивно. Ещё такую систему легко масштабировать, просто динамически подключая к сети новые компоненты, что делает peer-to-peer идеальным для распределённых вычислений, особенно в ситуациях, когда каждый участник никому ничего не должен и работает в сети только тогда, когда у него есть возможность.

Самый, пожалуй, известный пример peer-to-peer-системы — это BitTorrent, где каждый узел может работать как файловый сервер, но чем больше узлов, тем больше информации можно хранить и тем быстрее её можно скачивать. Есть и менее известные применения, например, сенсорные сети или стаи беспилотников, применяющиеся в военной или природоохранной сферах. Если один из беспилотников вышел из строя, стая переконфигурируется и продолжает выполнение задания.

Подробнее про Peer-to-peer мы поговорим, когда будем рассматривать архитектуру распределённых приложений.