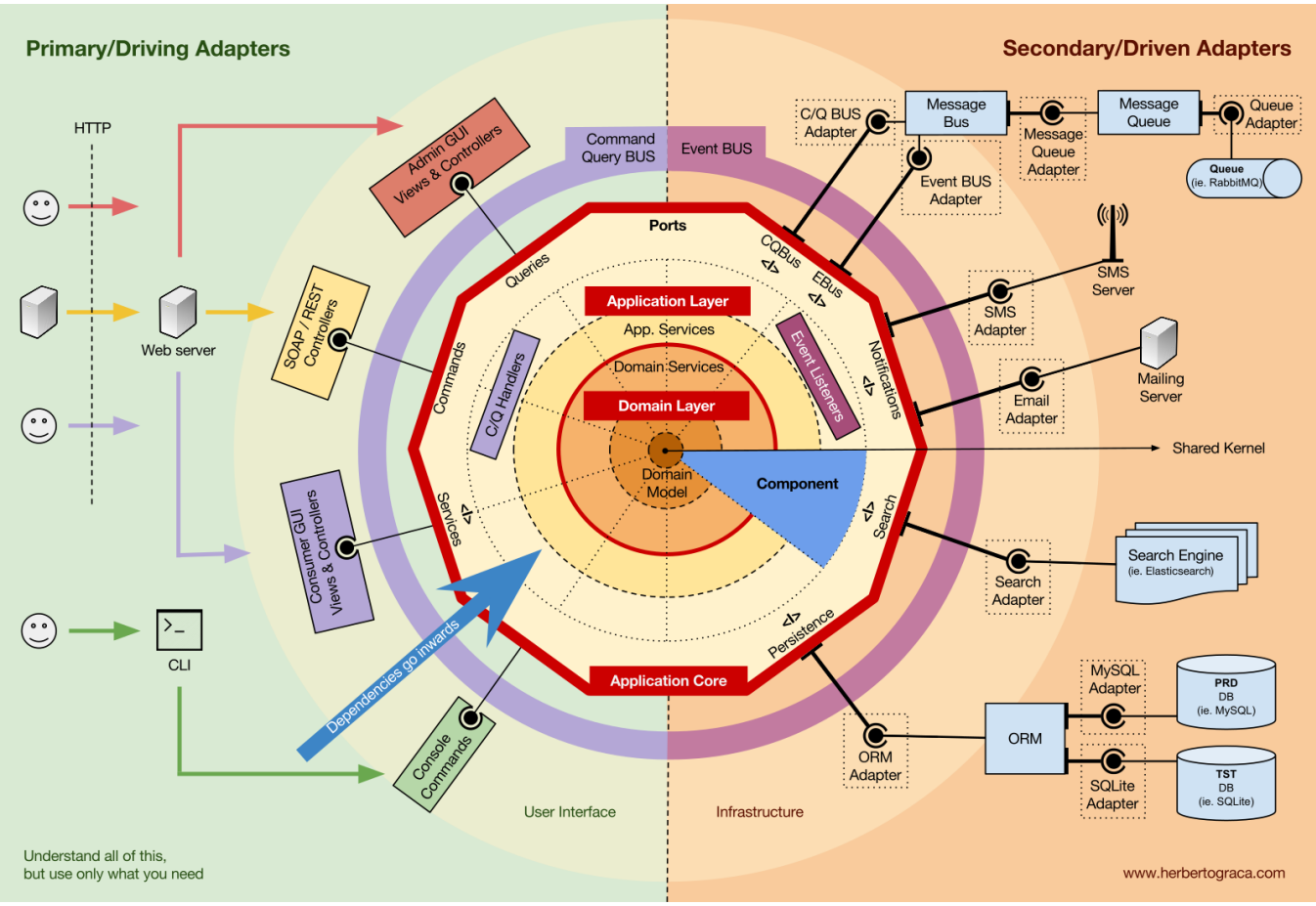
DDD, Hexagonal, Onion, Clean, CQRS… как я собрал всё это вместе

25 окт 2018 в 16:29



Эта статья — часть «Хроники архитектуры программного обеспечения», серии статей об архитектуре ПО. В них я пишу о том, что узнал об архитектуре программного обеспечения, что я думаю об этом и как использую знания. Содержание этой статьи может иметь больше смысла, если вы прочитаете предыдущие статьи в серии.

После окончания университета я начал работать учителем средней школы, но несколько лет назад уволился и пошёл в разработчики программного обеспечения на полный рабочий день.

С тех пор я всегда чувствовал, что мне нужно восстановить «потерянное» время и узнать как можно больше, как можно быстрее. Поэтому я стал немного увлекаться экспериментами, много читать и писать, уделяя особое внимание дизайну и архитектуре программного обеспечения. Вот почему я пишу эти статьи, чтобы помочь себе в обучении.

В последних статьях я рассказывал о многих концепциях и принципах, которые я узнал, и немного о том, как я рассуждаю о них. Но я представляю их как фрагменты одного большого паззла.

Эта статья о том, как я собрал воедино все эти фрагменты. Кажется, я должен дать им название, так что назову их **явной архитектурой** (explicit architecture). Кроме того, все эти концепции *«испытаны в бою»* и используются в продакшне на высоконадёжных платформах. Одна из них — SaaS-платформа электронной коммерции с тысячами интернет-магазинов по всему миру, другая — торговая площадка, работающая в двух странах с шиной сообщений, которая обрабатывает более 20 миллионов сообщений в месяц.

- Фундаментальные блоки системы
- Инструменты
- Подключение инструментов и механизмов доставки к ядру приложения
    - Порты
    - Основные или управляющие адаптеры
    - Вторичные или управляемые адаптеры
    - Инверсия управления

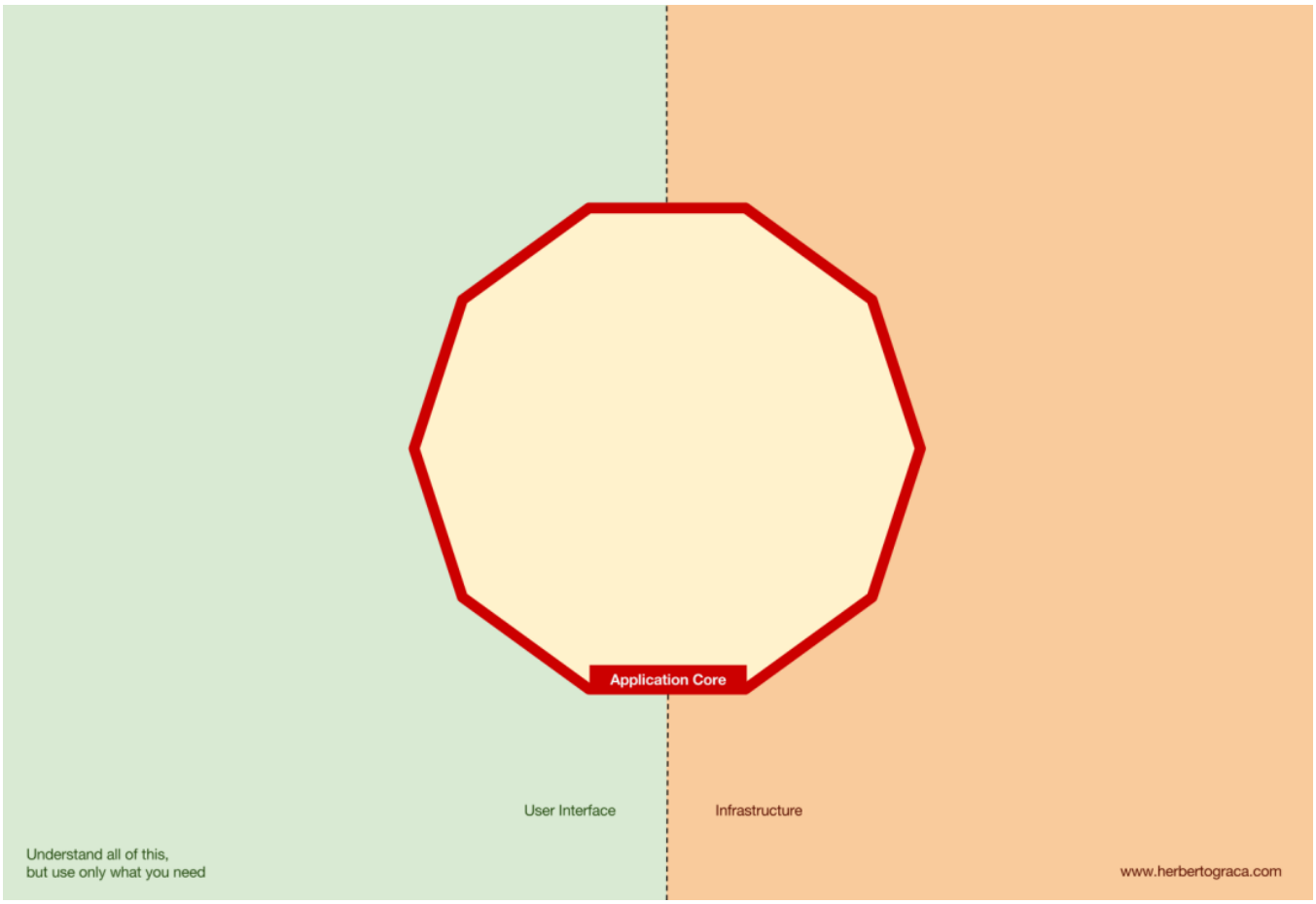
- Организация ядра приложения
  - Уровень приложения
  - Уровень домена
    - Сервисы домена
    - Модель домена
- Компоненты
  - Разъединение компонентов
    - Срабатывание логики в других компонентах
    - Получение данных от других компонентов
      - Общее хранилище данных для компонентов
      - Отдельное хранение данных для компонента
- Поток управления

Фундаментальные блоки системы

Начнём с того, что вспомним архитектуры EBI и Ports & Adapters. Обе они явно разделяют внутренний и внешний код приложения, а также адаптеры для соединения внутреннего и внешнего кода.

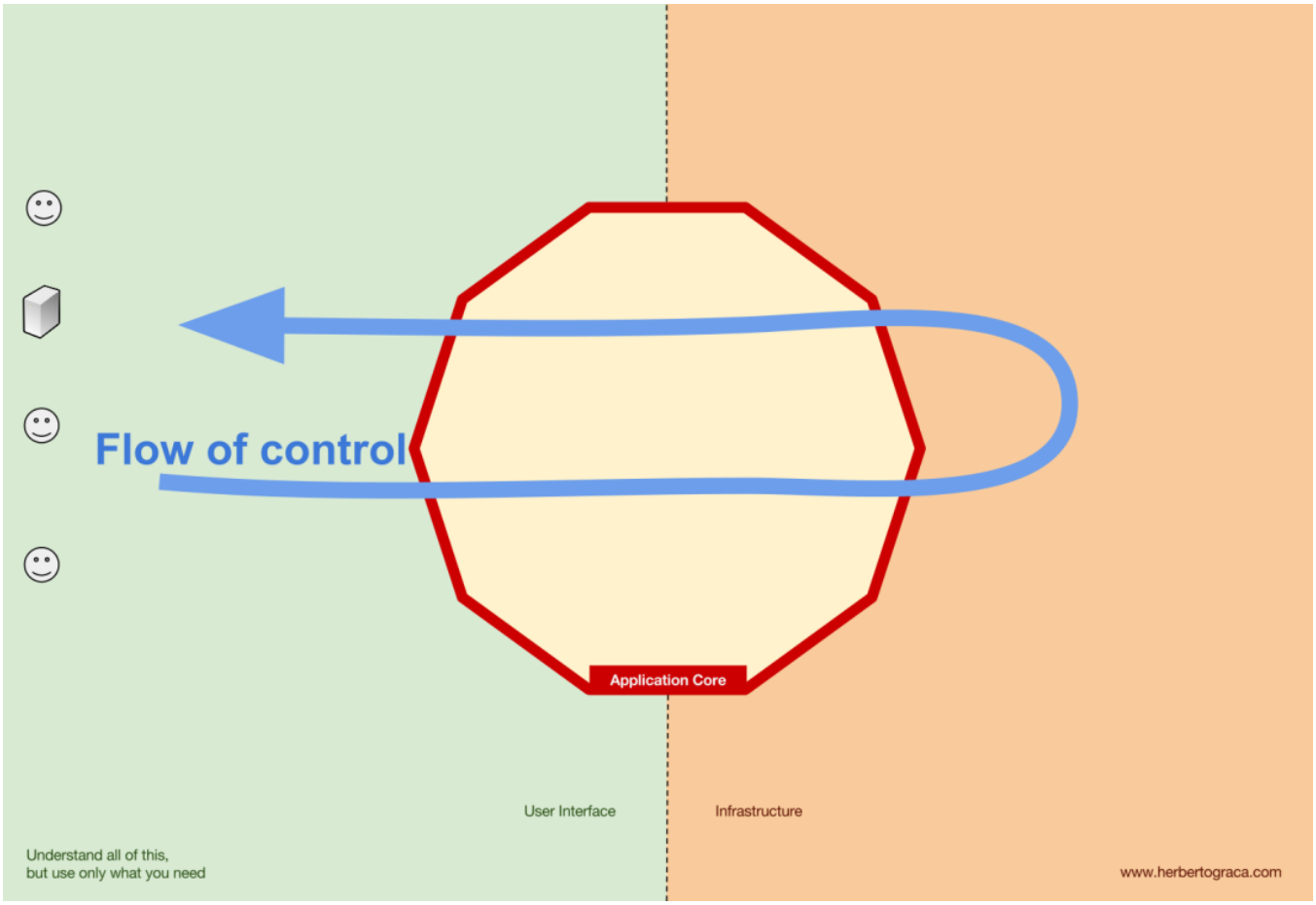
Кроме того, архитектура Ports & Adapters явно определяет три фундаментальных блока кода в системе:

- То, что позволяет запускать **пользовательский интерфейс**, независимо от его типа.
- Системная **бизнес-логика** или **ядро приложения**. Его использует UI для совершения реальных транзакций.
- Код **инфраструктуры**, который соединяет ядро нашего приложения с такими инструментами, как БД, поисковая система или сторонние API.



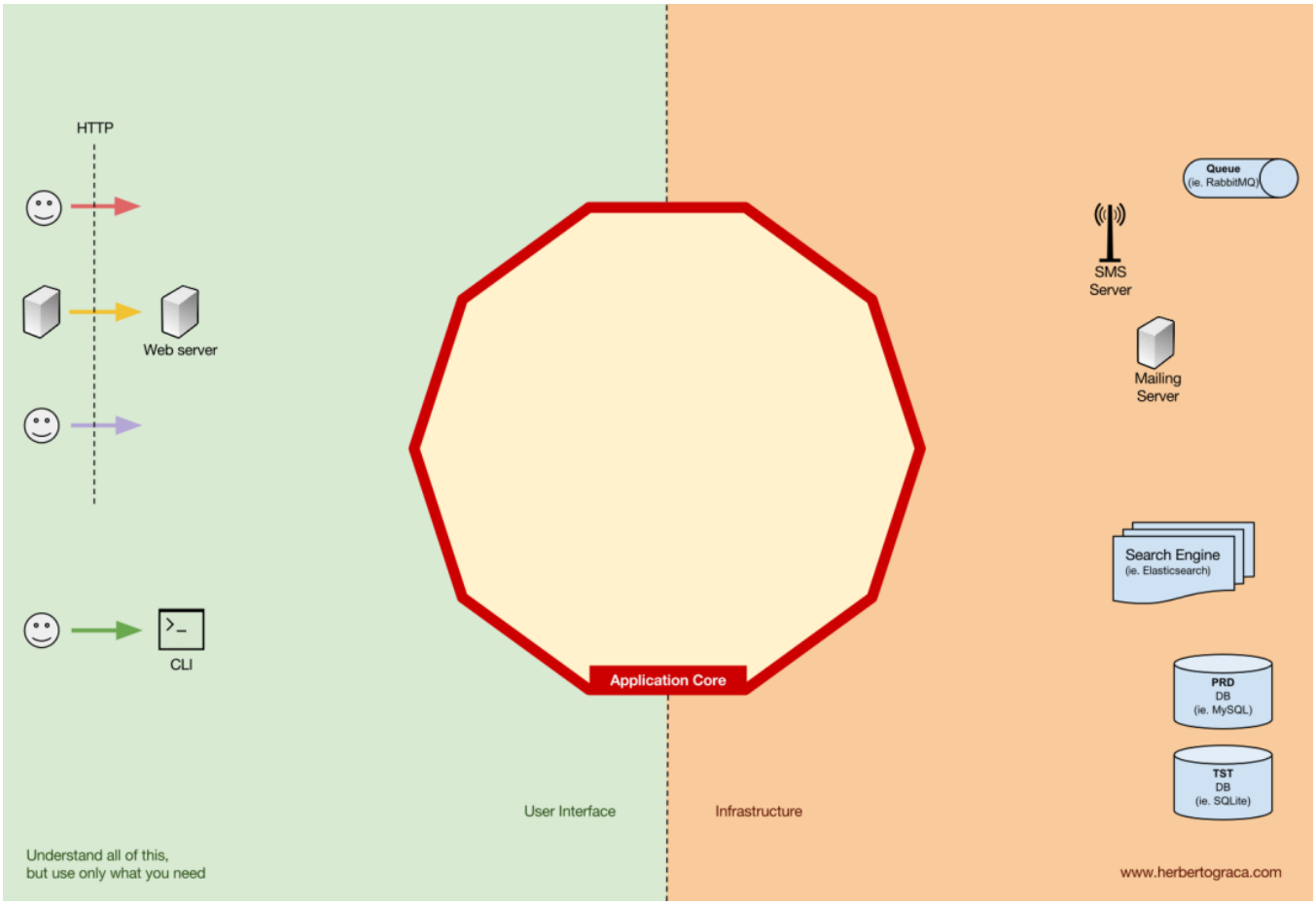
Ядро приложения — самое главное, о чём нужно думать. Этот код позволяет совершать реальные действия в системе, то есть это и ЕСТЬ наше приложение. С ним могут работать несколько пользовательских интерфейсов (прогрессивное веб-приложение, мобильное приложение, CLI, API и др.), всё выполняется на одном ядре.

Как можно себе представить, типичный поток выполнения идёт от кода в UI через ядро приложения к коду инфраструктуры, обратно к ядру приложения и, наконец, ответ доставляется в UI.



Инструменты

Вдали от самого важного кода ядра есть ещё инструменты, которые использует приложение. Например, ядро БД, поисковая система, веб-сервер и консоль CLI (хотя последние два также являются механизмами доставки).



Кажется странным отнести консоль CLI в тот же тематический раздел, что и СУБД, ведь у них разное предназначение. Но фактически и то, и другое — инструменты, используемыми приложением. Ключевое различие в том, что консоль CLI и веб-сервер **говорят приложению что-то сделать**, ядро СУБД, наоборот, **получает команды от приложения**. Это очень важное различие, поскольку оно сильно влияет на то, как мы пишем код для соединения этих инструментов с ядром приложения.

Подключение инструментов и механизмов доставки к ядру приложения

Блоки кода, соединяющие инструменты с ядром приложения, называются адаптерами (архитектура Ports & Adapters). Они позволяют бизнес-логике взаимодействовать с определённым инструментом, и наоборот.

Адаптеры, которые говорят приложению что-то делать, называются **первичными или управляющими адаптерами**, в то время как адаптеры, которым приложение говорит что-то делать, называются **вторичными или управляемыми адаптерами**.

Порты

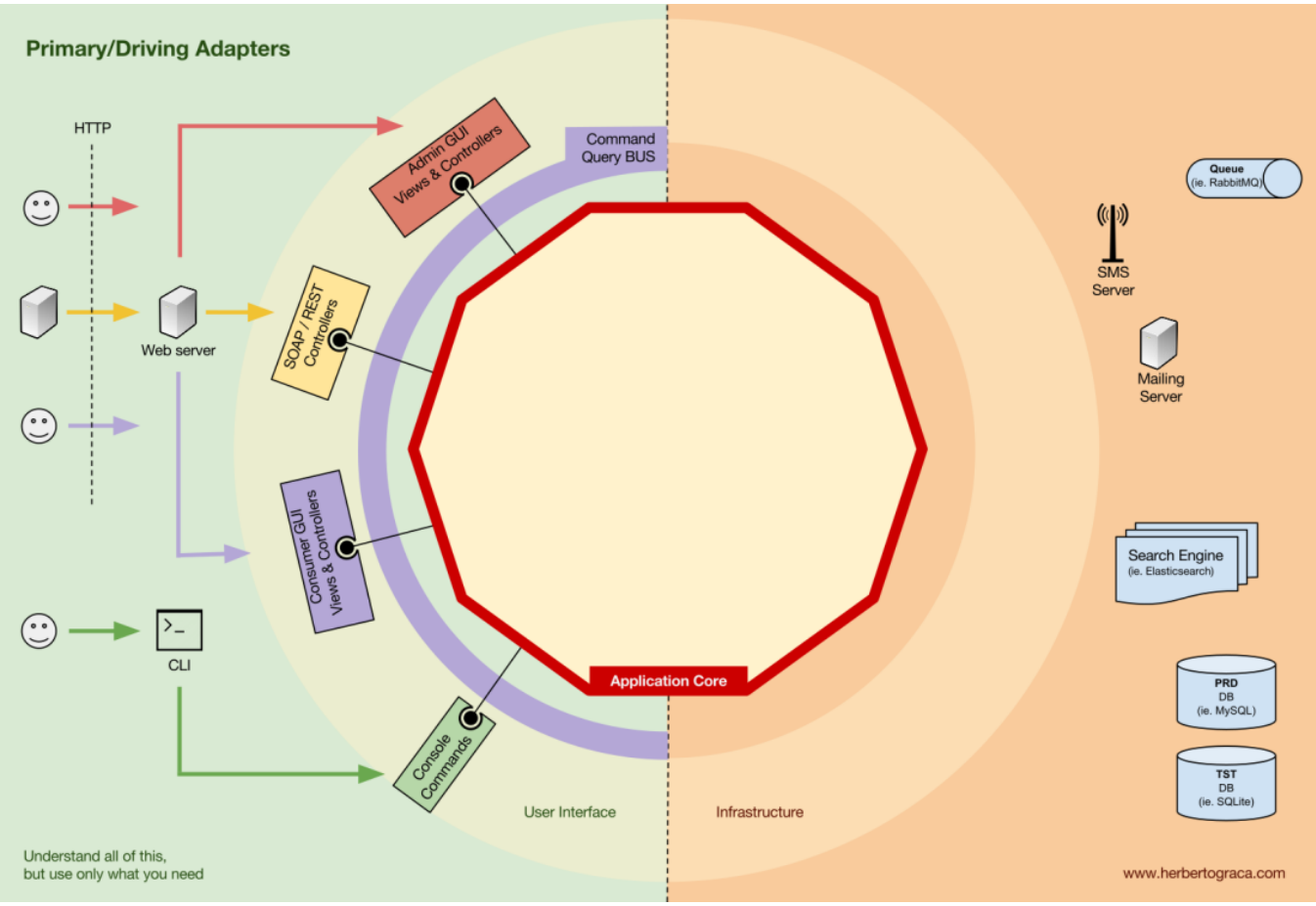
Однако эти *адаптеры* создаются не случайно, а чтобы соответствовать конкретной точке входа в ядро приложения, ***порту***. Порт — это **не более чем спецификация** того, как инструмент может использовать ядро приложения или наоборот. В большинстве языков и в простейшем виде этот порт будет интерфейсом, но фактически он может быть составлен из нескольких интерфейсов и DTO.

Важно отметить, что **порты (интерфейсы) находятся внутри бизнес-логики**, а адаптеры — снаружи. Чтобы этот шаблон работал должным образом, крайне важно создавать порты в соответствии с потребностями ядра приложения, а не просто имитировать API инструментов.

Основные или управляющие адаптеры

Основные или управляющие адаптеры **оборачиваются вокруг**

порта и используют его для указания ядру приложения, что делать. **Они преобразуют все данные от механизма доставки в вызовы методов в ядре приложения.**



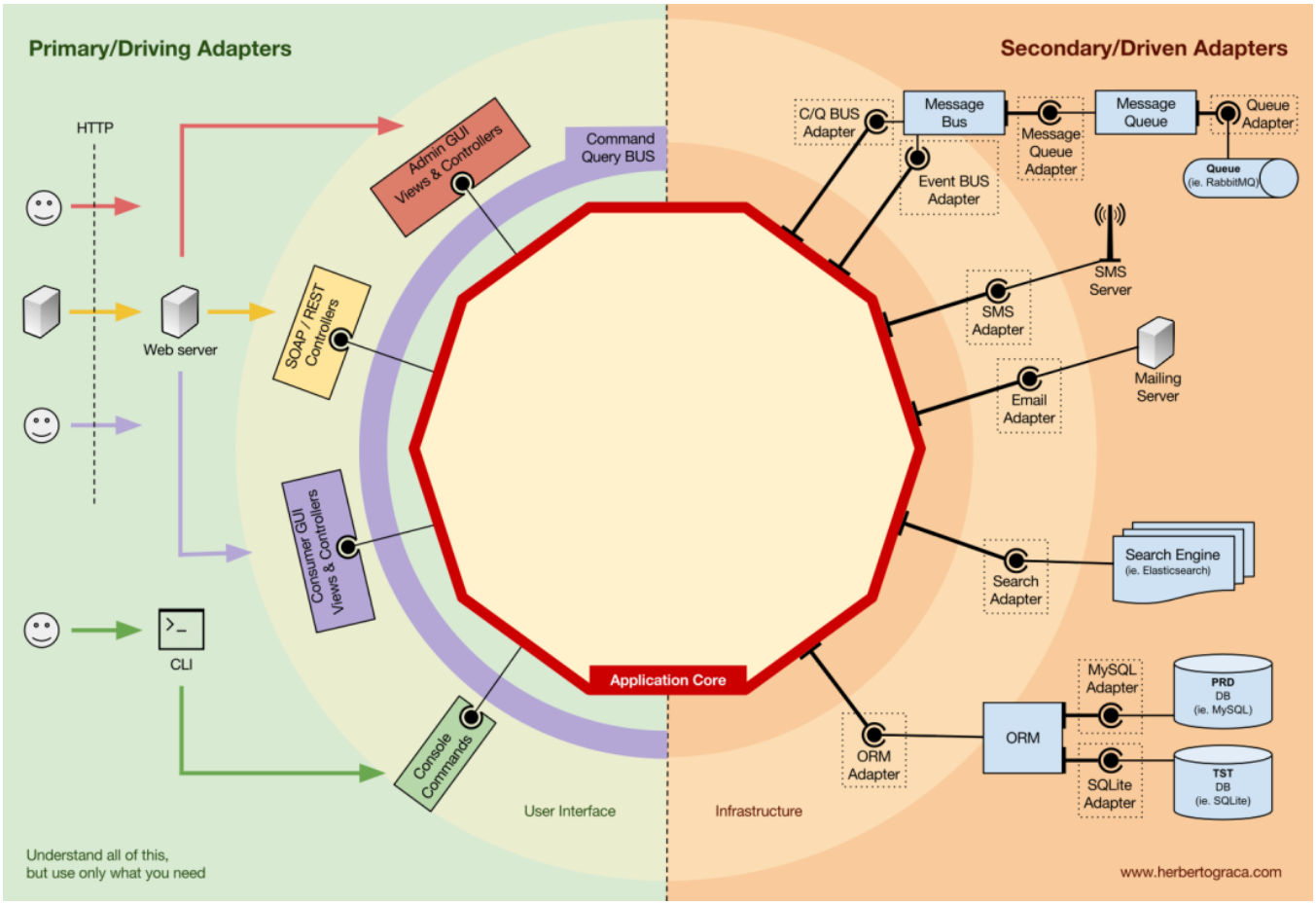
Другими словами, наши управляющие адаптеры являются контроллерами или консольными командами, они внедряются в их конструктор с некоторым объектом, класс которого реализует интерфейс (порт), который требует контроллер или консольная команда.

В более конкретном примере порт может быть интерфейсом службы или интерфейсом репозитория, который требуется контроллеру. Конкретная реализация сервиса, репозитория или запроса затем внедряется и используется в контроллере.

Кроме того, порт может быть шиной команд (command bus) или интерфейсом шины запросов (query bus). В этом случае конкретная реализация шины команд или запросов вводится в контроллер, который затем создает команду или запрос и передаёт его соответствующей шине.

Вторичные или управляемые адаптеры

В отличие от управляющих адаптеров, которые оборачиваются вокруг порта, **управляемые адаптеры** реализуют порт, интерфейс, а затем вводятся в ядро приложения там, где требуется порт (с указанием типа).



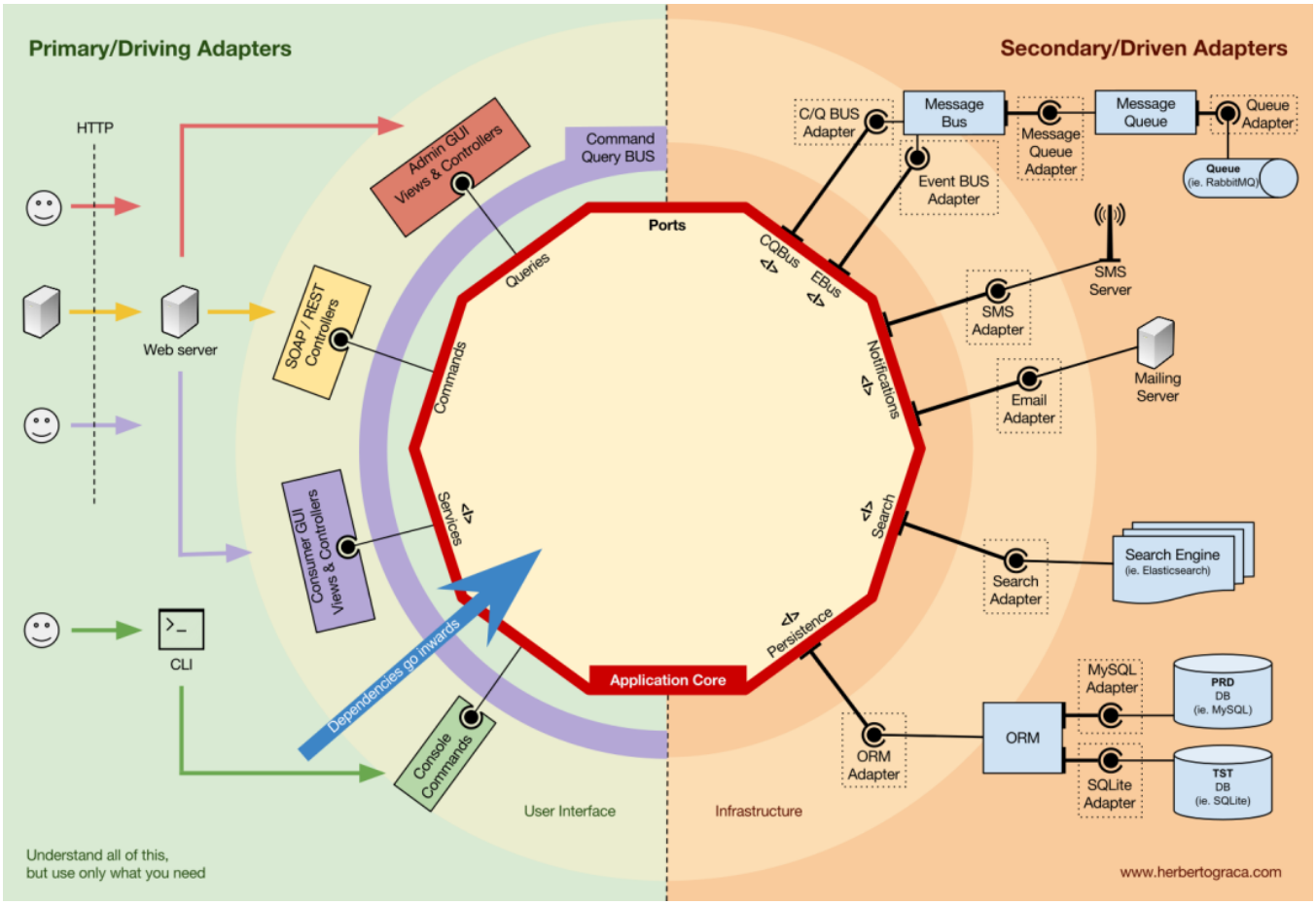
Например, у нас есть нативное приложение, которому необходимо сохранять данные. Мы создаём persistence-интерфейс с методом *сохранения* массива данных и методом *удаления* строки в таблице по её ID. С этого момента везде, где приложение должно сохранить или удалить данные, мы будем требовать в конструкторе объект, который реализует persistence-интерфейс, который мы определили.

Теперь создаём адаптер, специфичный для MySQL, который будет реализовывать этот интерфейс. У него будут методы для сохранения массива и удаления строки в таблице, и мы введём его везде, где требуется persistence-интерфейс.

Если в какой-то момент мы решим изменить поставщика базы данных, например, на PostgreSQL или MongoDB, нам просто нужно создать новый адаптер, который реализует persistence-интерфейс, специфичный для PostgreSQL, и внедрить новый адаптер вместо старого.

Инверсия управления

Характерной особенностью этого шаблона является то, что адаптеры зависят от конкретного инструмента и конкретного порта (путём реализации интерфейса). Но наша бизнес-логика зависит только от порта (интерфейса), который предназначен для удовлетворения потребностей бизнес-логики и не зависит от конкретного адаптера или инструмента.



Это означает, что зависимости направлены к центру, то есть налицо **инверсия принципа управления на архитектурном уровне**.

Хотя опять же, **крайне важно, чтобы порты создавались в соответствии с потребностями ядра приложения, а не просто имитировали API инструментов**.

Организация ядра приложения

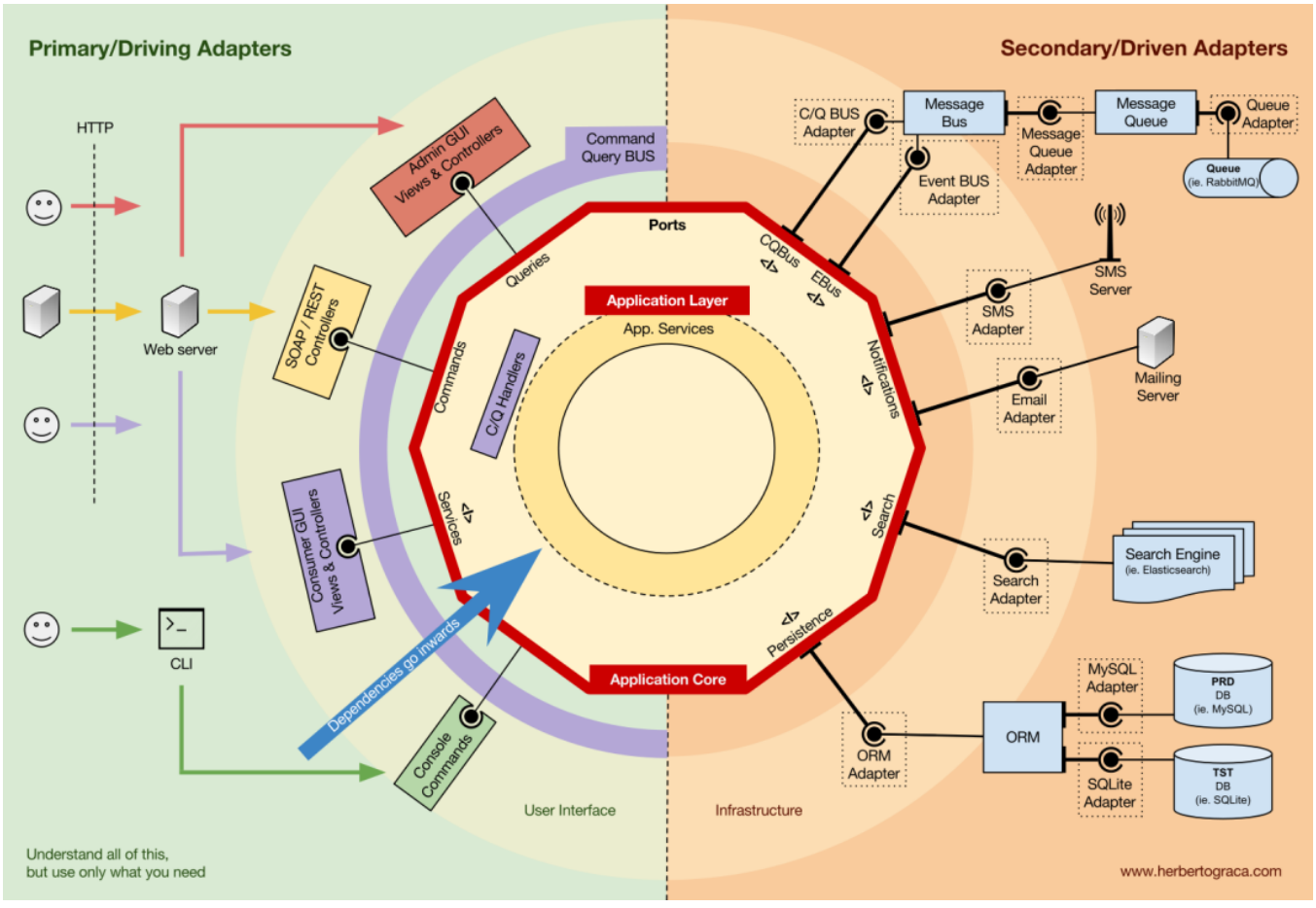
Архитектура Onion подхватывает слои DDD и включает их в архитектуру портов и адаптеров. Эти уровни предназначены для того, чтобы внести некоторый порядок в бизнес-логику, внутреннюю часть «шестиугольника» портов и адаптеров. Как и раньше,

направление зависимостей — к центру.

Уровень приложения (прикладной уровень)

Варианты использования — это процессы, которые можно запустить в ядре одним или несколькими пользовательскими интерфейсами. Например, в CMS может быть один UI для обычных пользователей, другой независимый UI для администраторов CMS, ещё один интерфейс CLI и веб-API. Эти UI (приложения) могут инициировать уникальные или общие варианты использования.

Варианты использования определяются на прикладном уровне — первом уровне DDD и архитектуры Onion.



Этот уровень содержит службы приложений (и их интерфейсы) в качестве объектов первого класса, а также содержит интерфейсы портов и адаптеров (порты), которые включают интерфейсы ORM, интерфейсы поисковых систем, интерфейсы обмена сообщениями и т. д. В случае, когда мы используем шину команд и/или шину запросов, на этом уровне находятся соответствующие обработчики команд и запросов.

Службы приложений и/или обработчики команд содержат логику развёртывания варианта использования, бизнес-процесс. Как правило, их роль следующая:

1. использовать репозиторий для поиска одной или нескольких сущностей;
2. попросить эти сущности выполнить некоторую логику домена;
3. и использовать хранилище, чтобы заново сохранить сущности, эффективно сохраняя изменения данных.

Обработчики команд можно использовать двумя способами:

1. Они могут содержать логику для выполнения варианта использования;
2. Их можно использовать как простые части соединения в нашей архитектуре, которые получают команду и просто вызывают логику, существующую в службе приложения.

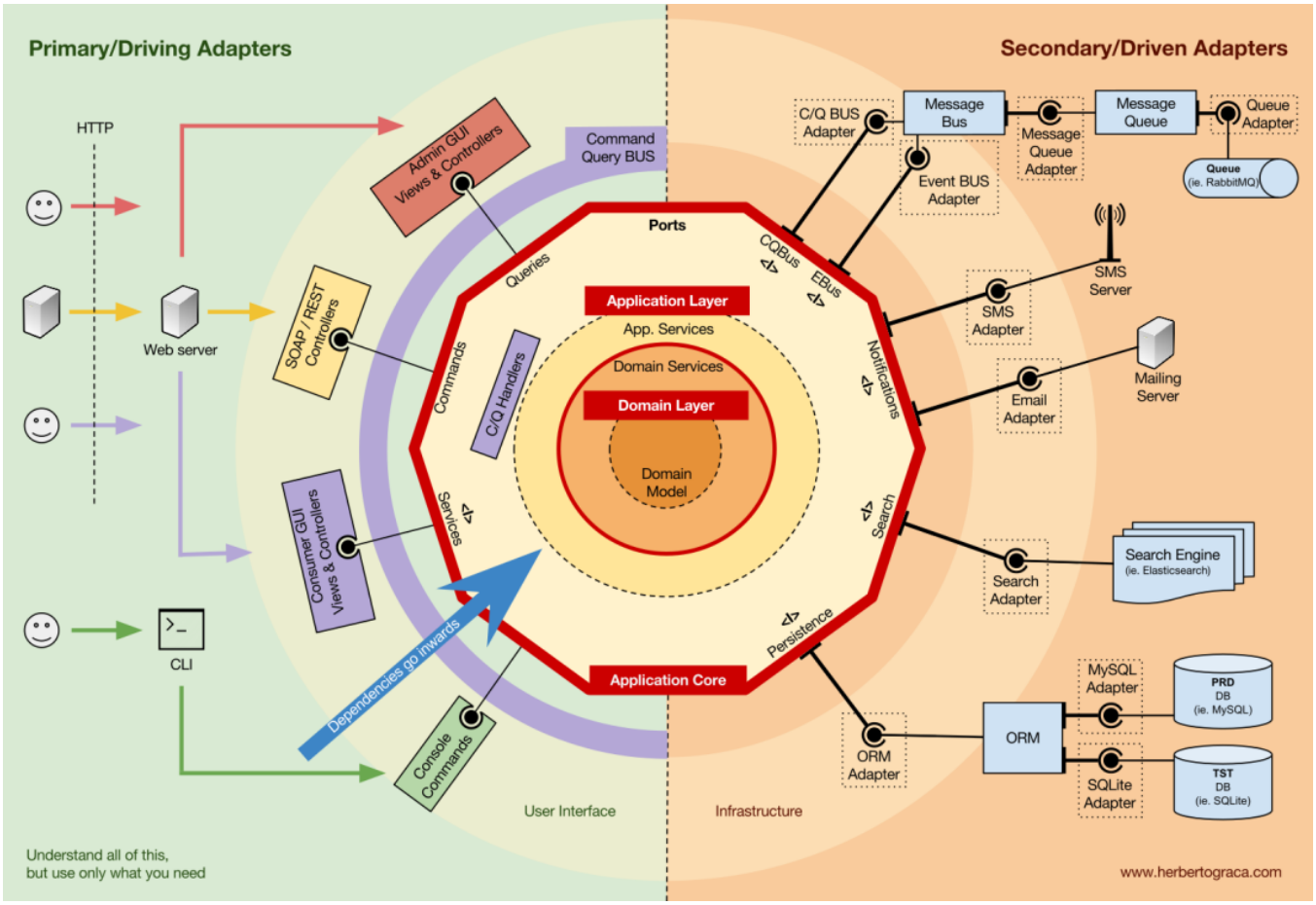
Какой подход использовать — зависит от контекста, например:

- У нас уже есть службы приложений и теперь добавляется шина команд?
- Позволяет ли шина команд указывать какой-либо класс/метод в качестве обработчика, либо необходимо расширить или реализовать существующие классы или интерфейсы?

Этот уровень также содержит инициирование **событий приложения**, которые представляют собой некоторый результат варианта использования. Эти события запускают логику, являющуюся побочным эффектом варианта использования, например, отправку сообщений электронной почты, уведомление стороннего API, отправку пуш-уведомления или даже запуск другого варианта использования, принадлежащего другому компоненту приложения.

Уровень домена

Далее внутри есть уровень домена. Объекты на этом уровне содержат данные и логику для управления этими данными, которые являются специфическими для самого домена и не зависят от бизнес-процессов, запускающих эту логику. Они независимы и совершенно не знают о прикладном уровне.



Сервисы домена

Как я уже упоминал выше, роль службы приложений:

1. использовать репозиторий для поиска одной или нескольких сущностей;
2. попросить эти сущности выполнить некоторую логику домена;
3. и использовать хранилище, чтобы заново сохранить сущности, эффективно сохраняя изменения данных.

Но иногда мы сталкиваемся с некоторой доменной логикой, которая

включает в себя различные сущности одинакового или разных типов, и эта доменная логика не принадлежит самим сущностям, то есть логика не является их прямой ответственностью.

Поэтому нашей первой реакцией может быть размещение этой логики вне сущностей в службе приложений. Однако это означает, что в других случаях логика домена не будет использоваться повторно: логика домена должна оставаться вне уровня приложения!

Решение состоит в том, чтобы создать доменную службу, роль которой состоит в получении набора сущностей и выполнении на них некоторой бизнес-логики. Доменная служба принадлежит к доменному уровню и поэтому ничего не знает о классах на прикладном уровне, таких как службы приложений или репозитории. С другой стороны, она может использовать другие доменные службы и, конечно же, объекты модели домена.

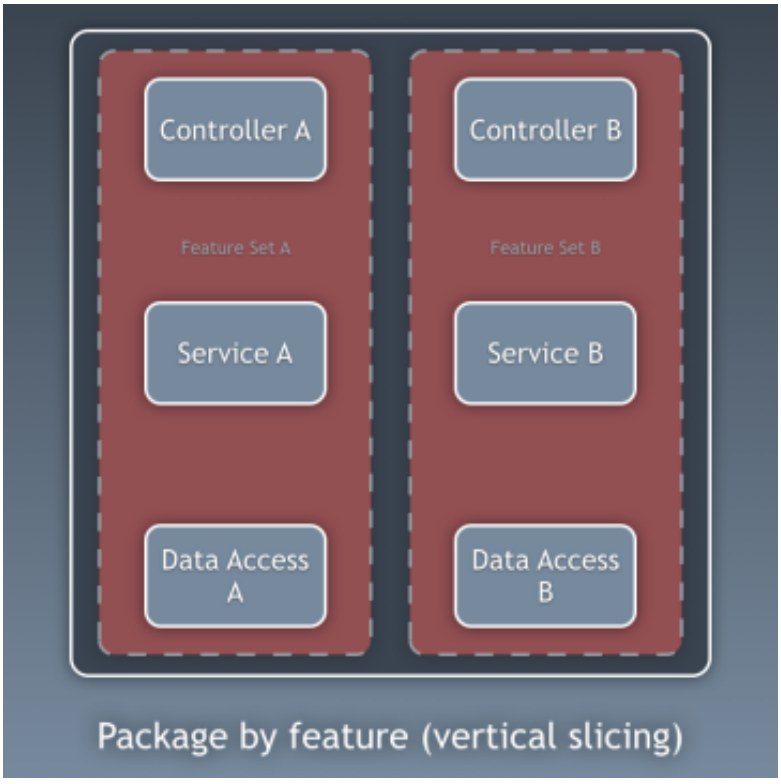
Модель домена

В самом центре находится модель домена. Она не зависит ни от чего за пределами этого круга и содержит бизнес-объекты, что-то представляющие в домене. Примерами таких объектов являются, прежде всего, сущности, а также объекты-значения (value objects), перечисления (enums) и любые объекты, используемые в модели домена.

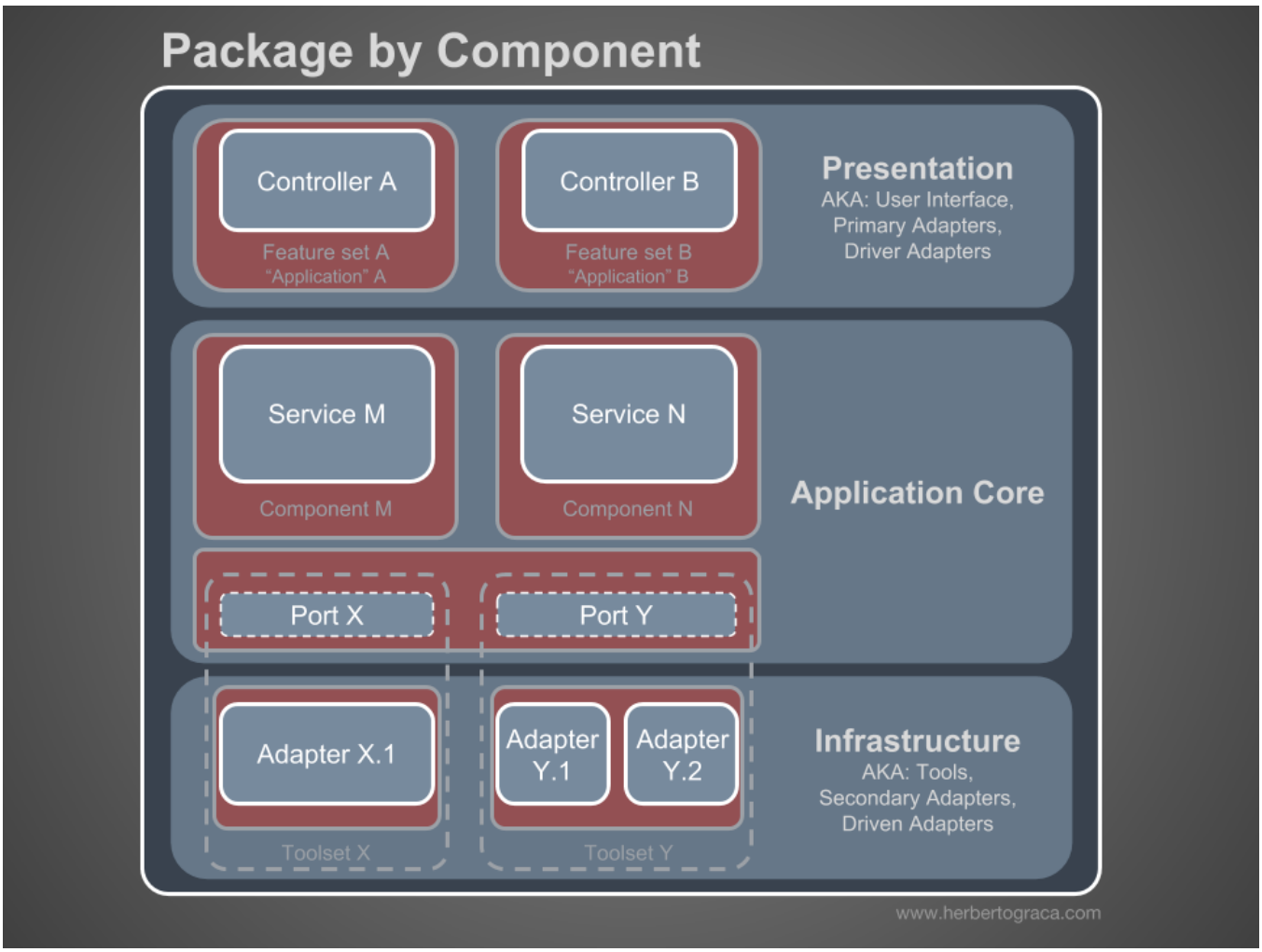
В модели домена «живут» также события домена. Когда определённый набор данных изменяется, то инициируются эти события, которые содержат новые значения изменённых свойств. Эти события идеально подходят, например, для использования в модуле регистрации событий (event sourcing).

Компоненты

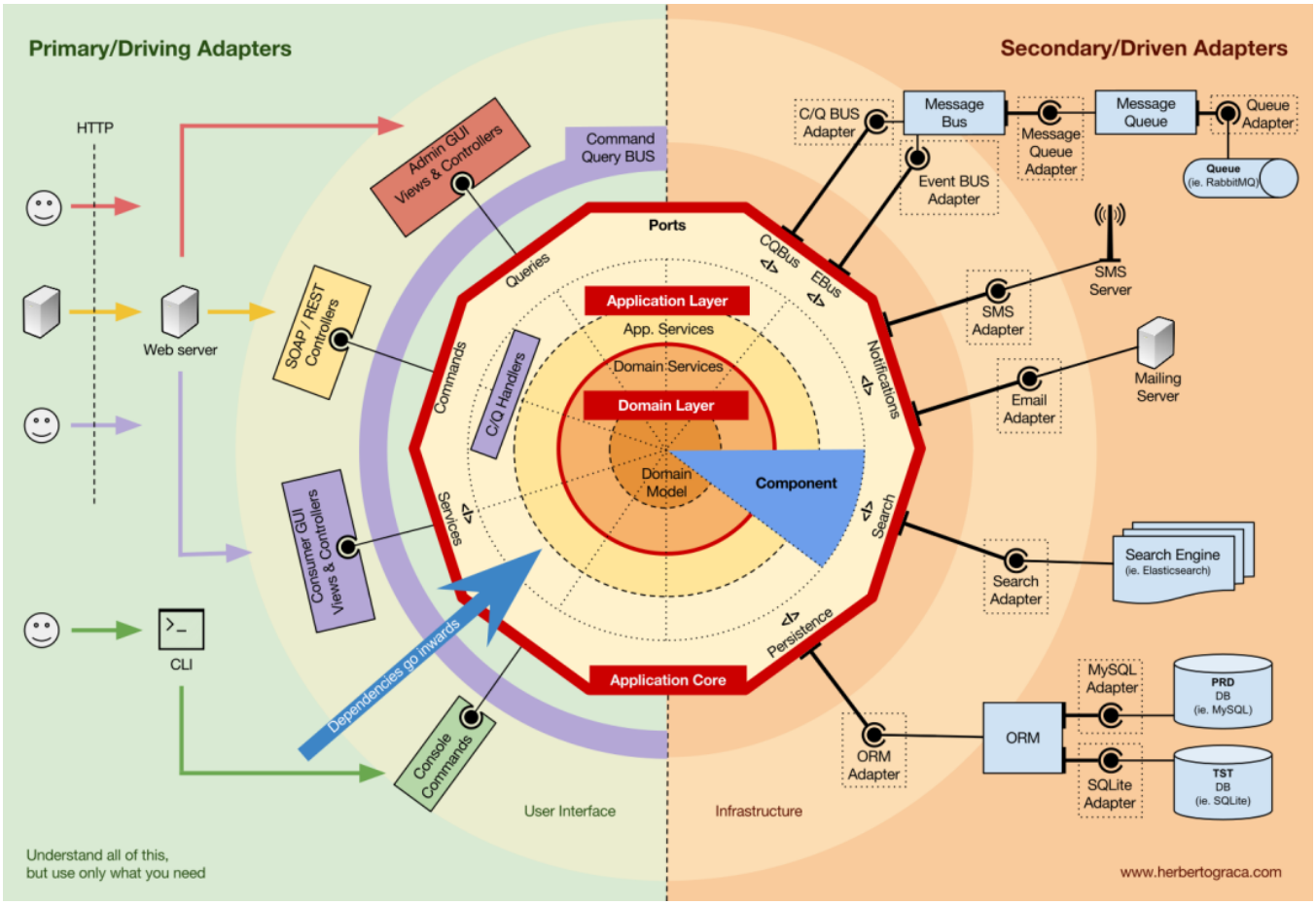
До сих пор мы изолировали код по слоям, но это слишком подробная изоляция кода. Не менее важно посмотреть на картину более общим взглядом. Речь идёт о разделении кода по поддоменам и ***связанным контекстам*** в соответствии с идеями Роберта Мартина, выраженными в ***screaming-архитектуре*** [то есть архитектура должна «кричать» о самом приложении, а не о том, какие фреймворки в нём используются — прим. пер.]. Здесь говорят об организации пакетов по функциям или по компонентам, в не по слоям, и это довольно хорошо объяснил Саймон Браун в статье «Пакеты по компонентам и тестирование в соответствии с архитектурой» в своём блоге:



Я сторонник организации пакетов по компонентов и хочу бесстыдно изменить диаграмму Саймона Брауна следующим образом:



Эти разделы кода сквозные для всех слоёв, описанных ранее, и это ***компоненты*** нашего приложения. Примеры компонентов — биллинг, пользователь, проверка или учётная запись, но они всегда связаны с доменом. Ограниченные контексты, такие как авторизация и/или аутентификация, должны рассматриваться как внешние инструменты, для которых мы создаём адаптер и прячемся за каким-то портом.

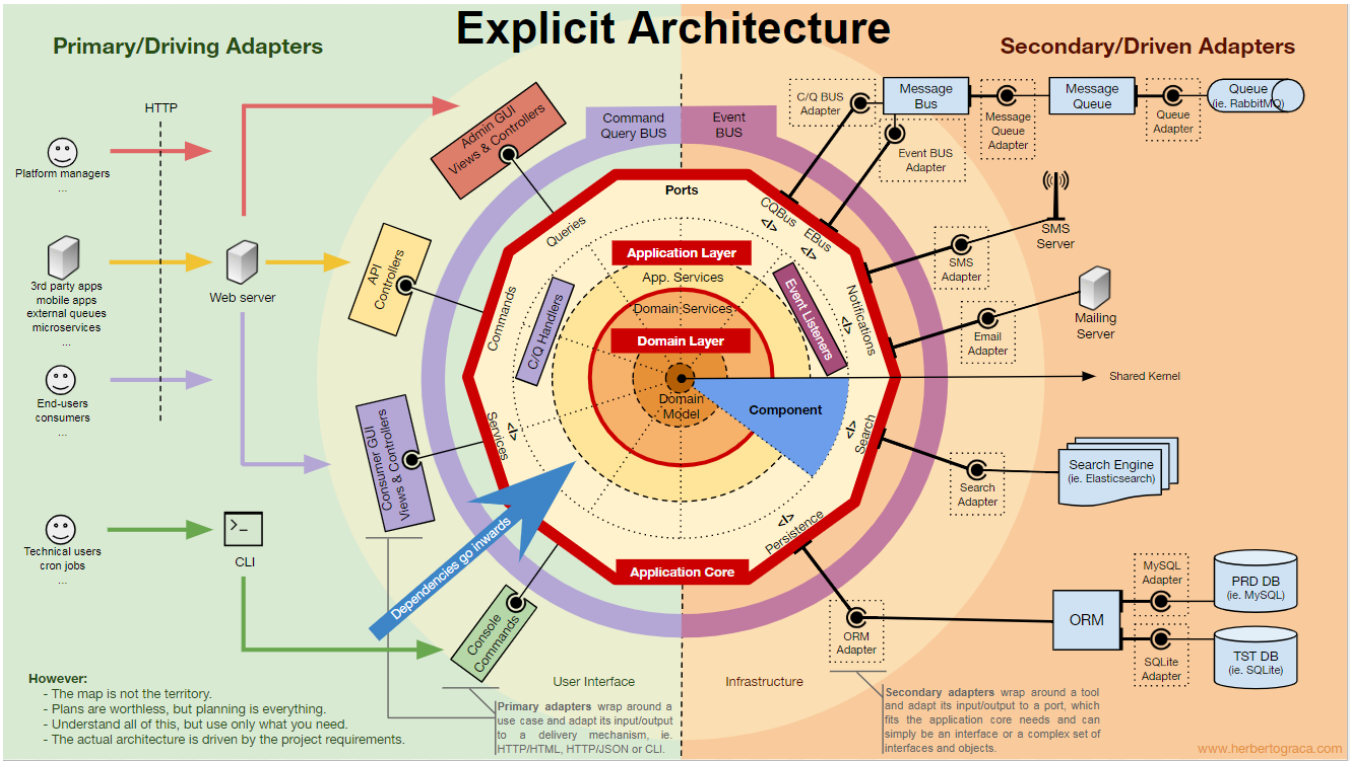


Разъединение компонентов

Так же, как в мелкозернистых единицах кода (классы, интерфейсы, трейты, миксины и проч.), крупные единицы (компоненты) выигрывают от слабого сцепления и плотной связности.

Чтобы разъединить классы, мы используем инъекцию зависимостей, вводя зависимости в класс, а не создавая их внутри класса, а также инверсию зависимостей, делая класс зависимым от абстракций (интерфейсов и/или абстрактных классов) вместо конкретных классов. Это означает, что зависимый класс ничего не знает о конкретном классе, который будет использовать, у него нет ссылки на полное имя классов, от которых он зависит.

Точно так же в полностью разъединенных компонентах каждый компонент ничего не знает ни о каком другом компоненте. Другими словами, у него нет ссылки на какой-либо мелкозернистый блок кода из другого компонента, даже на интерфейс! Это означает, что инъекции зависимостей и инверсии зависимостей недостаточно для разделения компонентов, нам понадобятся какие-то архитектурные конструкции. Могут понадобиться события, общее ядро, согласованность в конечном счёте (eventual consistency) и даже служба обнаружения сервисов (discovery service)!



Срабатывание логики в других компонентах

Когда один из наших компонентов (компонент B) должен что-то делать всякий раз, когда что-то ещё происходит в другом

компоненте (компонент A), мы не можем просто сделать прямой вызов из компонента A в класс/метод компонента B, потому что тогда A будет связан с B.

Однако мы можем использовать диспетчер событий для отправки события приложения, которое будет доставлено любому компоненту, прослушивающему его, включая B, и прослушиватель событий в B вызовет желаемое действие. Это означает, что компонент A будет зависеть от диспетчера событий, но будет отделён от компонента B.

Тем не менее, если само событие «живёт» в A, это означает, что B знает о существовании A и связан с ним. Чтобы удалить эту зависимость, мы можем создать библиотеку с набором функциональных возможностей ядра приложения, которые будут совместно использоваться всеми компонентами — общее ядро. Это означает, что оба компонента будут зависеть от общего ядра, но будут отделены друг от друга. Общее ядро содержит функциональные возможности, такие как события приложения и домена, но оно также может содержать объекты спецификации и всё, что имеет смысл совместно использовать. При этом оно должно быть минимального размера, поскольку любые изменения в общем ядре повлияют на все компоненты приложения. Кроме того, если у нас есть polyglot-система, скажем, экосистема микросервисов на разных языках, то общее ядро не должно зависеть от языка, чтобы его понимали все компоненты. Например, вместо общего ядра с классом событий оно будет содержать описание события (то есть имя, свойства, возможно, даже методы, хотя они были бы более полезны в объекте спецификации) на универсальном языке вроде JSON, чтобы все компоненты/микросервисы могли интерпретировать его и, возможно, даже автоматически генерировать свои собственные конкретные реализации.

Этот подход работает как в монолитных, так и в распределённых приложениях, таких как экосистемы микросервисов. Но если события можно доставить только асинхронно, то этого подхода недостаточно для контекстов, где логика запуска в других компонентах должна срабатывать немедленно! Здесь компоненту A потребуется делать прямой HTTP-вызов к компоненту B. В этом случае, чтобы разъединить компоненты, нам понадобится служба обнаружения. Компонент A спросит у неё, куда отправить запрос, чтобы инициировать желаемое действие. Как вариант, сделать запрос к службе обнаружения, которая передаст его соответствующей службе и в конечном счёте возвратит ответ инициатору запроса. Этот подход связывает компоненты со службой обнаружения, но не связывает их друг с другом.

Получение данных от других компонентов

Как я вижу, компоненту не разрешено изменять данные, которыми он не «владеет», но он может запрашивать и использовать любые данные.

Общее хранилище данных для компонентов

Если компонент должен использовать данные, принадлежащие другому компоненту (например, компонента биллинга должен использовать имя клиента, которое принадлежит компоненту аккаунтов), то он содержит объект запроса к хранилищу этих данных. То есть компонент биллинга может знать о любом наборе данных, но должен использовать «чужие» данные только для чтения.

Отдельное хранение данных для компонента

В этом случае применяется тот же шаблон, но уровень хранения данных становится сложнее. Наличие компонентов с собственным хранилищем данных означает, что каждое хранилище данных содержит:

- Набор данных, которыми компонент владеет и может изменить, сделав его единственным источником истины;
- Набор данных, который является копией данных других компонентов, которые он не может изменить сам по себе, но они необходимы для функциональности компонента. Эти данные должны обновляться всякий раз, когда они изменяются в компоненте владельца.

Каждый компонент создаст локальную копию необходимых ему данных из других компонентов, которые будут использоваться по необходимости. При изменении данных в компоненте, которому они принадлежат, этот компонент владельца инициирует событие домена, несущее изменения данных. Компоненты, содержащие копию этих данных, будут прослушивать это событие домена и соответствующим образом обновлять свою локальную копию.

Поток управления

Как я уже сказал выше, поток управления идёт от пользователя в ядро приложения, к инструментам инфраструктуры, затем опять в ядро приложения — и обратно к пользователю. Но как именно

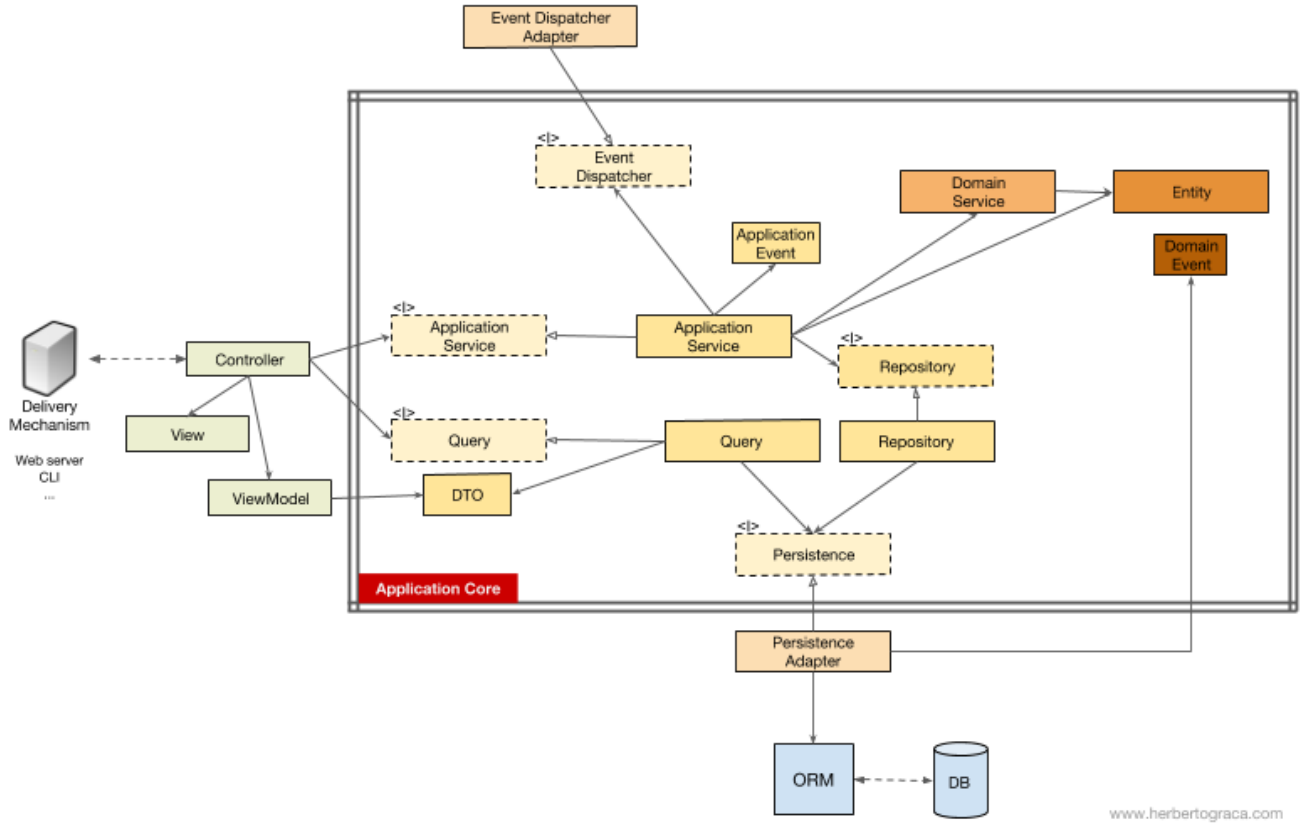
классы работают вместе? Кто от кого зависит? Как мы их составляем?

Как Дядя Боб в своей статье о «чистой» архитектуре (Clean Architecture), я постараюсь объяснить поток управления схемами UMLish…

Без шины команд/запросов

Если мы не используем шину команд, контроллеры будут зависеть либо от службы приложения, либо от объекта Query.

[Дополнение 18.11.2017] Я полностью пропустил DTO, который использую для возврата данных из запроса, поэтому сейчас добавил его. Спасибо MorphineAdministered, который указал на пробел.



На приведённой выше диаграмме мы используем интерфейс для службы приложений, хотя можем утверждать, что он на самом деле не нужен, так как служба приложений является частью нашего кода приложения. Но мы не хотим менять реализацию, хотя и можем провести полный рефакторинг.

Объект Query содержит оптимизированный запрос, который просто возвращает некоторые необработанные данные, которые будут показаны пользователю. Эти данные возвращаются в DTO, который внедрён в ViewModel. Эта ViewModel может иметь какую-то логику View и будет использоваться для заполнения View.

С другой стороны, служба приложений содержит логику вариантов использования, которая срабатывает, когда мы хотим что-то сделать в системе, а не просто просмотреть некоторые данные. Служба приложений зависит от репозиториев, которые возвращают сущности, содержащие логику, которую необходимо инициировать. Она может также зависеть от доменной службы для координации процесса домена в нескольких сущностях, но это редкий случай.

После разбора случая использования, сервис приложения может уведомить всю систему, что произошёл случай использования, тогда он будет зависеть ещё от диспетчера событий, чтобы инициировать событие.

Интересно отметить, что мы размещаем интерфейсы как на persistence-движке, так и на репозиториях. Это может показаться излишним, но они служат разным целям:

- Persistence-интерфейс является уровнем абстракции над ORM, так что мы можем поменять ORM без изменений в ядре

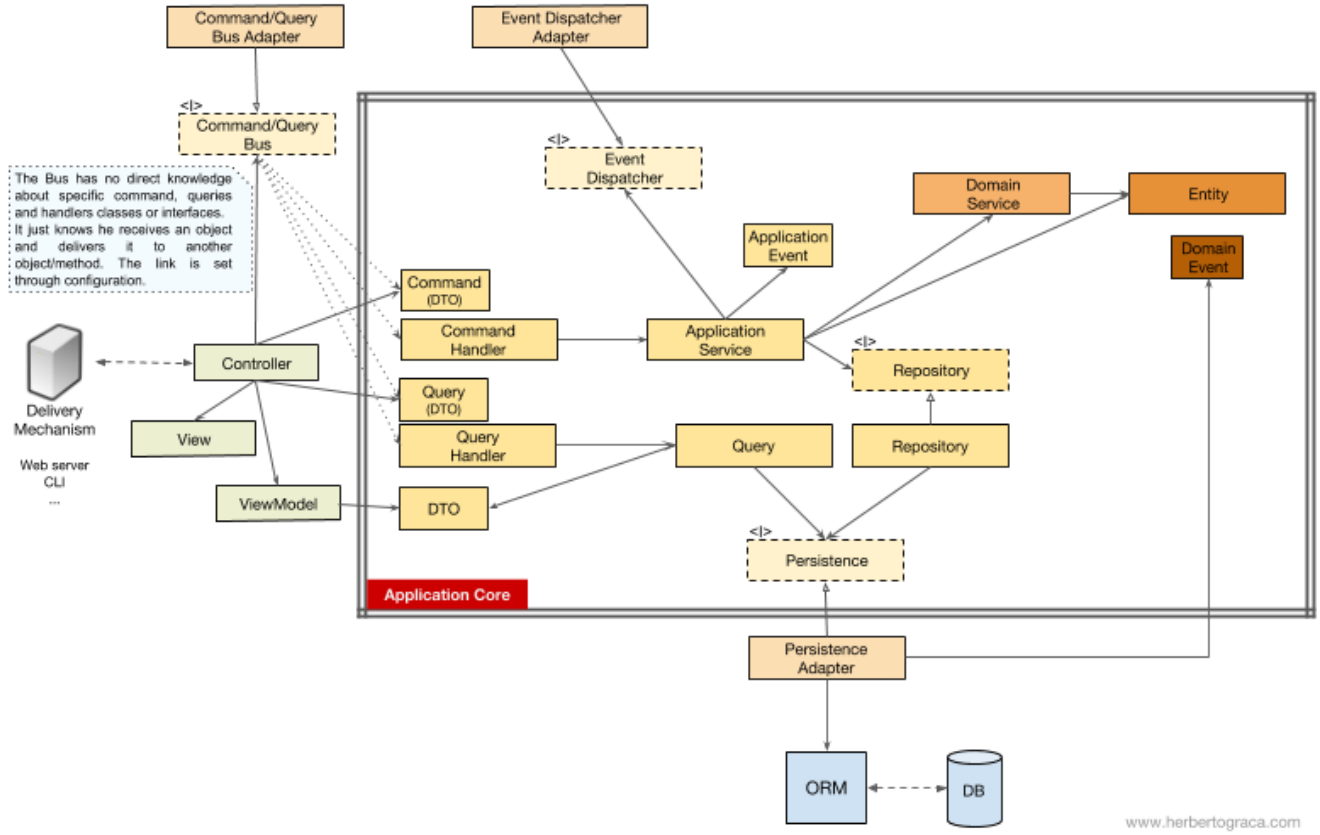
приложения.
- Интерфейс репозитория является абстракцией над самим persistence-движком. Допустим, мы хотим переключиться с MySQL на MongoDB. В этом случае persistence-интерфейс может остаться тем же, а если мы хотим продолжить использовать тот же ORM, даже адаптер сохраняемости останется тем же самым. Тем не менее, язык запросов совершенно другой, поэтому мы можем создавать новые репозитории, которые используют один и тот же механизм сохранения, реализуют одни и те же интерфейсы репозитория, но строят запросы, используя язык запросов MongoDB вместо SQL.

С шиной команд/запросов

В случае, если наше приложение использует шину команд/запросов, диаграмма остаётся практически такой же, за исключением того, что контроллер теперь зависит от шины, а также от команд или запросов. Здесь создаётся экземпляр команды или запроса и передаётся шине, которая найдёт соответствующий обработчик для получения и обработки команды.

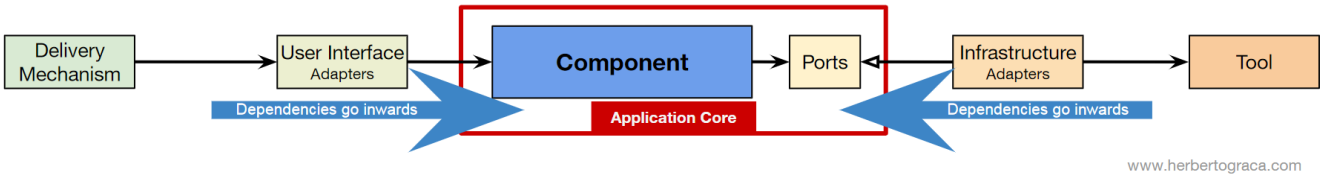
На приведённой ниже схеме обработчик команд использует службу приложений. Но это не всегда необходимо, ведь в большинстве случаев обработчик будет содержать всю логику варианта использования. Нам нужно лишь извлечь логику из обработчика в отдельную службу приложений, если требуется повторно использовать ту же логику в другом обработчике.

[Дополнение 18.11.2017] Я полностью пропустил DTO, который использую для возврата данных из запроса, поэтому сейчас добавил его. Спасибо MorphineAdministered, который указал на пробел.



Вы могли заметить, что нет никаких зависимостей между шиной, командой, запросом и обработчиками. По сути, они не должны знать друг о друге, чтобы обеспечить хорошее разъединение. Способ направления шины на конкретный обработчик для обработки команды или запроса настраивается в простой конфигурации.

В обоих случаях все стрелки — зависимости, которые пересекают границу ядра приложения — указывают внутрь. Как объяснялось ранее, это фундаментальное правило архитектуры Ports & Adapters, архитектуры Onion и чистой архитектуры Clean.



Заключение

Как всегда, цель состоит в том, чтобы получить разъединённую кодовую базу с высокой связностью, в которой можно легко, быстро и безопасно производить любые изменения.

> Планы бесполезны, но планирование — это всё. — *Эйзенхауэр*

Эта инфографика — карта концептов. Знание и понимание всех этих концептов помогает спланировать здоровую архитектуру и работоспособное приложение.

Однако:

> Карта — это не территория. — *Альфред Корзыбский*

Другими словами, **это всего лишь рекомендации! Приложение — это территория, реальность, конкретный вариант использования, где нужно применить наши знания, и именно оно определяет, как будет выглядеть настоящая архитектура!**

Мы должны понимать все эти закономерности, но также всегда нужно думать и понимать, что именно нужно нашему приложению, как далеко мы можем зайти ради разъединения и связности. Это решение зависит от множества факторов, начиная с функциональных требований проекта, до сроков разработки

приложения, срока его службы, опыта команды разработчиков и так далее.

Вот так я всё это для себя представляю.

Эти идеи немного подробнее рассматриваются в следующей статье: «Больше, чем просто концентрические слои».