DDD, Hexagonal, Onion, Clean, CQRS… Как я объединил их вместе

Перевод статьи: DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together (Herberto Graca)

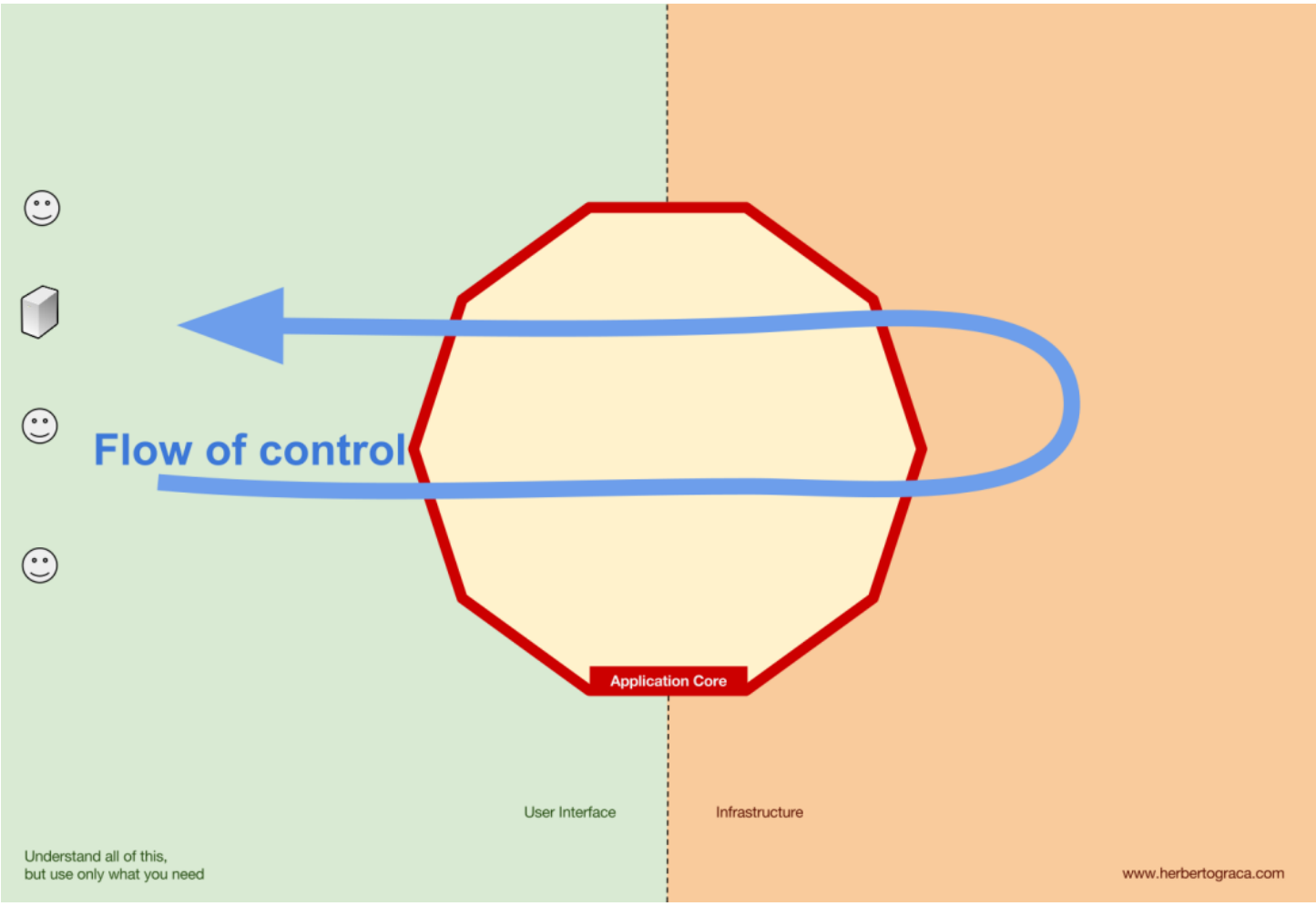
В этой статье я попытался объединить лучшие архитектурные практики, принципы и шаблоны вместе. То что в результате получилось я назвал **явной архитектурой** (Explicit Architecture).

Фундаментальные блоки системы

Я начну с упоминания EBI и Ports & Adapters (Hexagonal) архитектур. В обоих архитектурах имеет место разделение на внутреннюю часть приложения, внешнюю часть и часть которая необходима для соединения этих двух частей.

Более того, Ports & Adapters (Hexagonal) архитектура определяет 3 фундаментальных части системы:

- Часть, которая позволяет использовать интерфейс пользователя (UI) независимо от типа UI
- Бизнес-логика (ядро приложения), которая используется интерфейсом пользователя
- Инфраструктура: БД, поисковые системы, сторнние API и т.п.

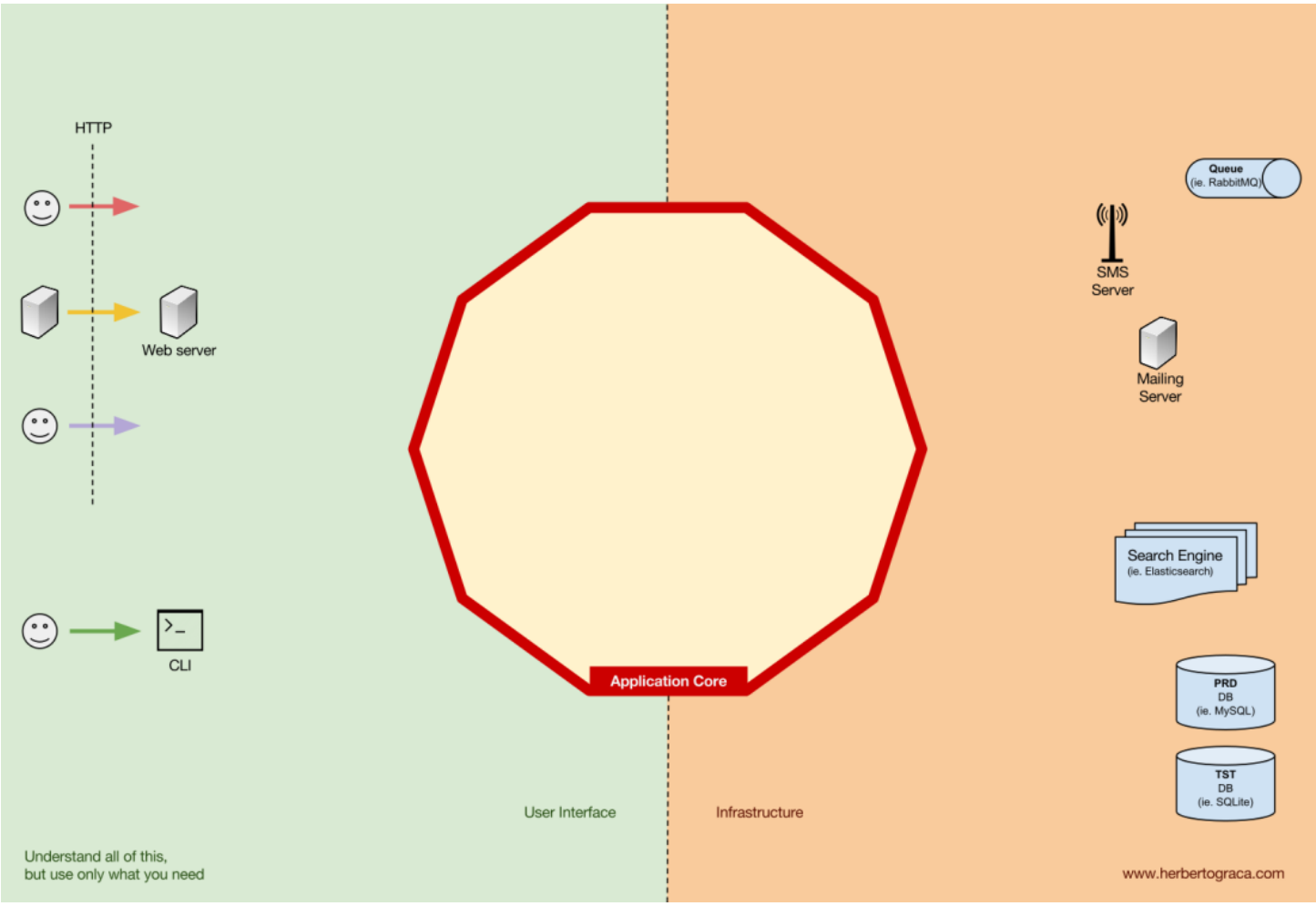


Ядро приложения (Application Core) – это то, о чем мы действительно должны заботиться. Оно может быть использовано разными UI (Web-интерфейс, мобильное приложение, CLI, API, ...).

Как вы можете себе представить, типичный поток управления идет от кода в пользовательском интерфейсе, через ядро приложения к коду инфраструктуры, обратно к ядру приложения и, наконец, выводит ответ на пользовательский интерфейс.

Инструменты

Подальше от ядра приложения у нас находятся инструменты (Tools), которые используются нашим приложением, например, БД, поисковый механизм, Web-сервер или CLI-консоль (последние 2 являются механизмами доставки).



Может показаться странным поместить CLI-консоль в ту же “корзину”, что и БД, при том, что они имеют разные назначение, на самом деле они являются инструментами, используемыми приложением. Основное разничие в том, что CLI-консоль и Web-сервер используются для того, чтобы **попросить наше приложение сделать что-либо**, тогда как БД используются для того, чтобы **выполнять запросы нашего приложения**. Это очень важное различие, поскольку оно сильно влияет на то, как мы создаем код, который связывает инструменты с ядром приложения.

Подключение инструментов и механизмов доставки к ядру приложения

Объекты, которые соединяют инструменты и ядро пиложения называются **адаптерами** (Adapters в Ports & Adapters (Hexagonal) архитектуре). Они реализуют код, который позволяет бизнес-логике коммуницировать с инструментами и наоборот.

Адаптеры, которые просят приложение сделать что-либо называются **Перевичными или Управляющими Адапртерами** (**Primary or Driving Adapters**), в то время как те адаптеры, которые используются для выполнения запросов приложения называются **Вторичными или Управляемыми Адаптерами** (**Secondary or Driven Adapters**).

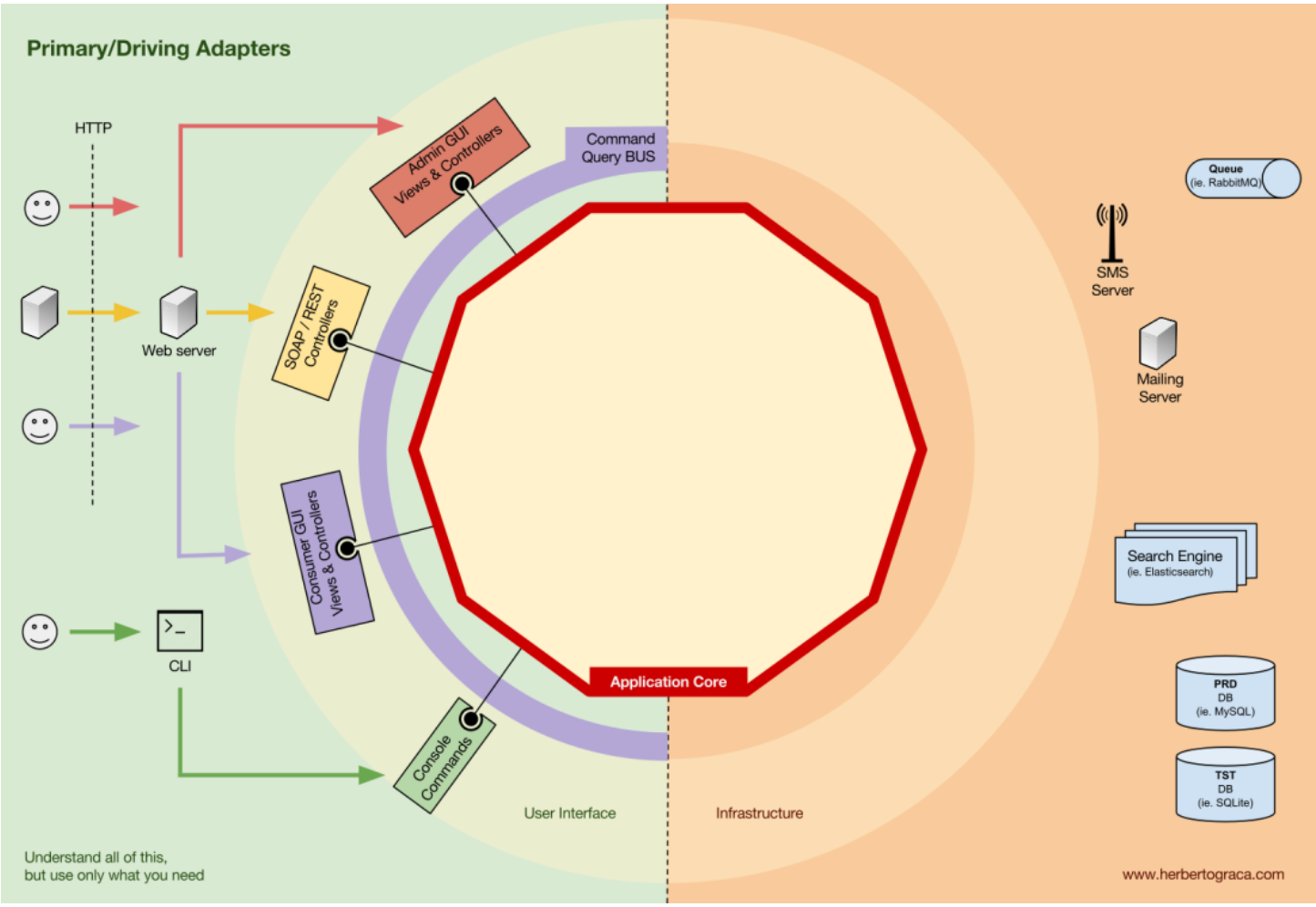
Порты

Адаптеры не создаются случайным образом. Они созданы для того чтобы соответствовать очень специфическим точка входа в ядро приложения, **Портам**. **Порт** – это не более чем спецификация, отражающая то, как инструмент может использовать ядро приложения или как ядро приложения может использовать инструмент. В большинстве языком программирования такую спецификацию отражает **интерфейс**, но также порт может выражаться в виде нескольких интерфейсов и DTO.

Очень важно отметить, что потры (интерфейсы) расспологаются внутри ядра приложения, тогда как адаптеры расспологаются снаружи. И чтобы этот шаблон работал должным образом, крайне важно, чтобы порты создавались в соответствии с потребностями ядра приложения, а не просто имитировали API инструментов.

Первичные или Управляющие Адаптеры

Первичные или Управляющие Адапреты содержат порт и используют его, чтобы сказать ядру приложения, что необходимо сделать. Они транслируют то, что пришло от механизма доставки (например, http-запрос через web-сервер) в вызов метода порта (интерфейса, определнного в ядре приложения).



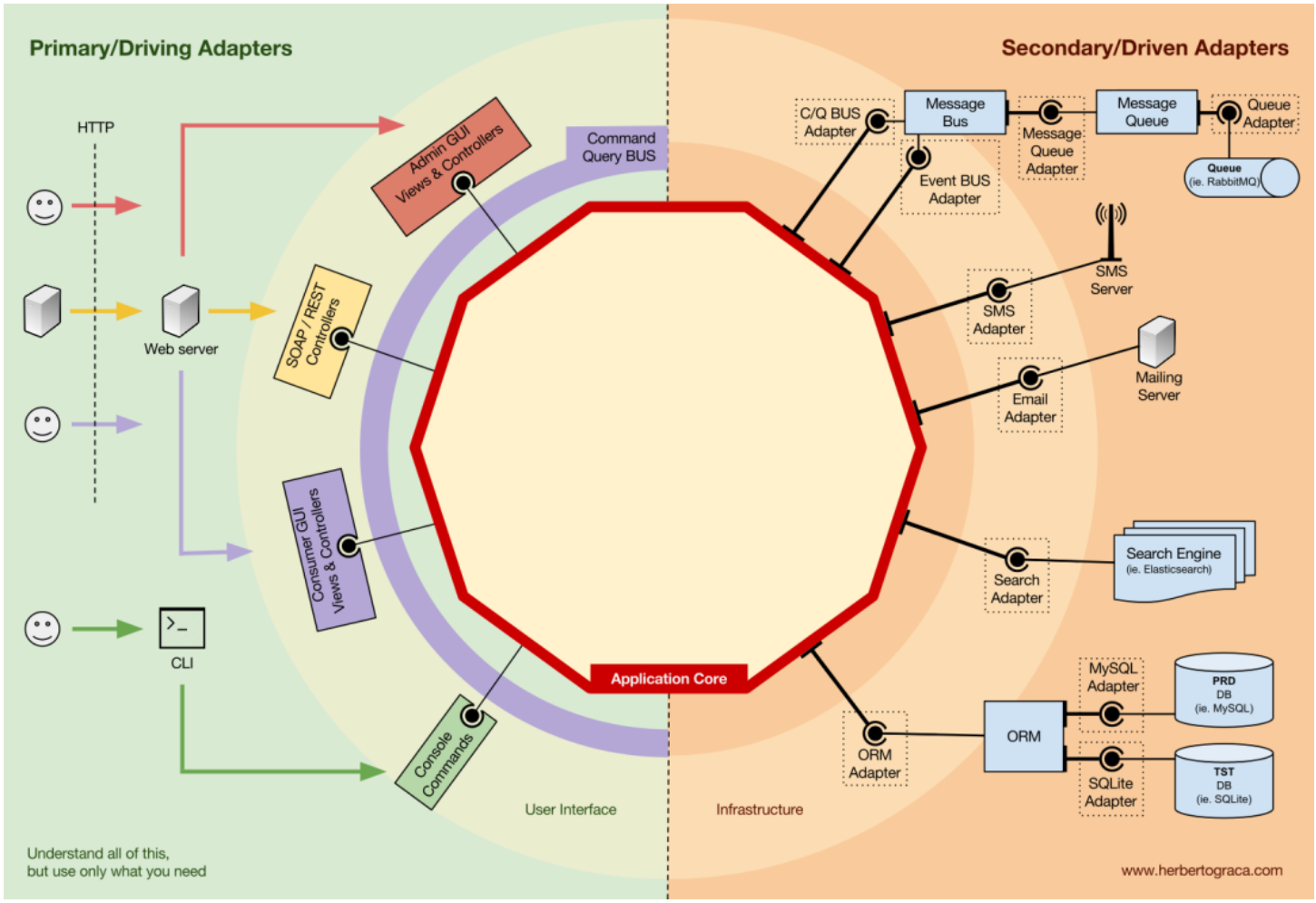
Другими словами, Управляющие Адаптеры – это Controller'ы или Console Command'ы, в которые внедряется (через конструктор или любым другим способом) объект, который реализует интерфейс (порт), который необходим Controller'у или Console Command'е для взаимодействия с ядром приложения.

Более конкретным примером может служить интерфейс Application Service'а (или Use Case'а) или интерфейс Repository'я. Далее конкретная реализаци Service'а, Repository или Query внедряется и используется, например, в Controller'е.

Также портом может быть интерфейс Command Bus'а или Query Bus'а. В этом случае конкретная реализация Command Bus'а или Query Bus'а внедряется в Controller, который далее создает Command или Query и передает его соответствующему Bus'у.

Вторичные или Управляемые Адаптеры

В отличии от Управляющих Адаптеров, которые содержат порт, **Управляемые Адаптеры <u>реализуют</u> порт** (интерфейс) и далее внедряются в объекты ядра приложения, туда где этот порт необходим.



Например, предположим, что у нас есть приложение, которое должно сохранять данные. Поэтому мы создаем интерфейс (порт), который для этого необходим, например, с методом для ***сохранения*** массива данных и методом для ***удаления*** строки в таблице БД по ID. С этого момента везде, где наше

приложение должно сохранять или удалять данные, мы будем требовать в своем конструкторе объект (управляемый адаптер), который реализует данный интерфейс.

Теперь мы создаем адаптер, специфичный для MySQL, который реализует этот интерфейс. Он будет иметь методы для сохранения массива и удаления строки в таблице, и мы будем вводить его везде, где он требуется.

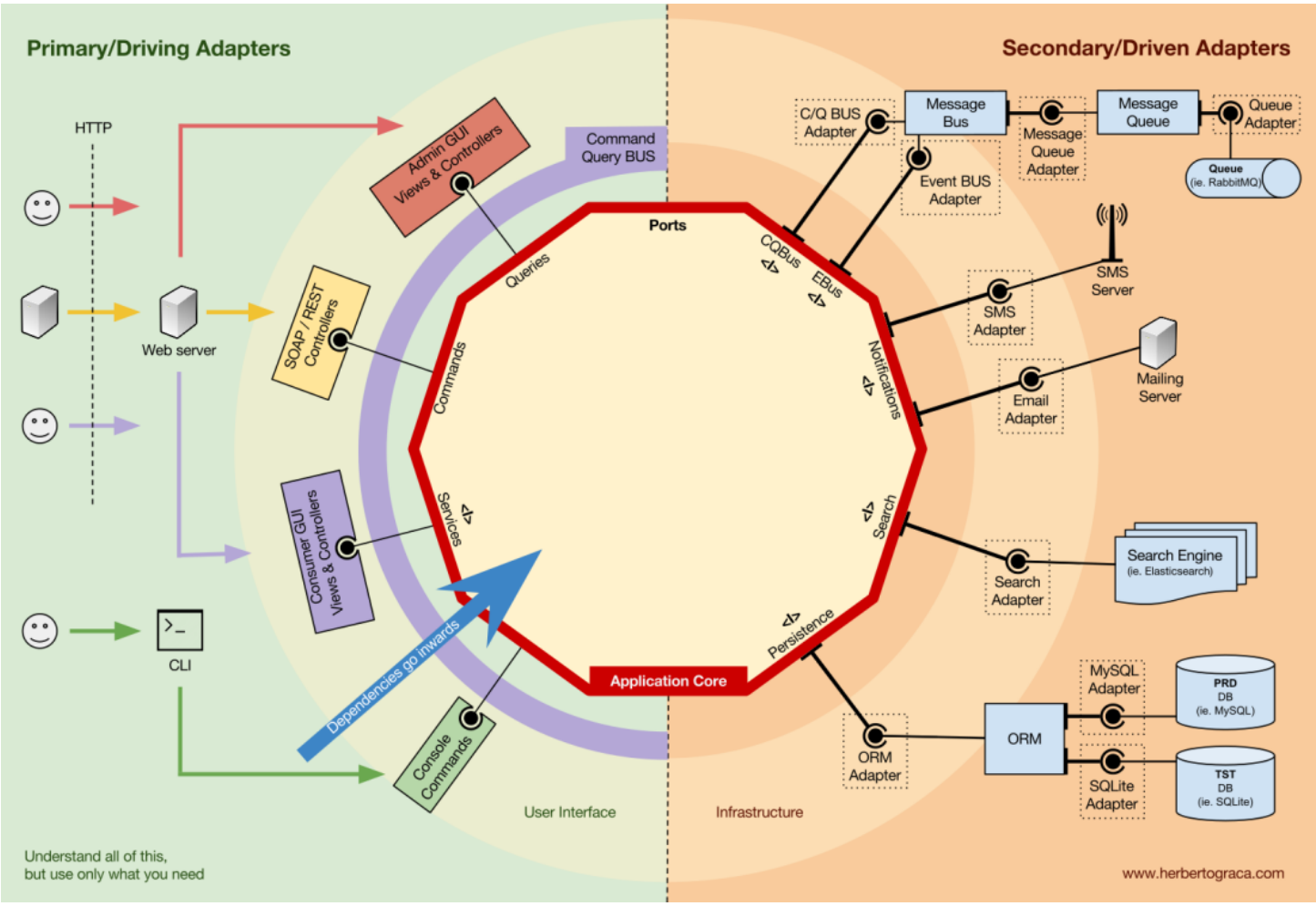
Если в какой-то момент мы решим сменить поставщика БД, например, на PostgreSQL или MongoDB, то нам будет необходимо только создать адаптер, который реализует наш интфейс (порт) и спецефичен для, например, PostgreSQL, и далее внедрить новый адаптер вместо старого.

Инверсия зависимостей

Необходимо отметить, что адаптер зависит от конкретного инструмета и от конкретного порта (по средствам реализации интерфейса). Но наша бизнес-логика (которая находится в ядре приложения) зависит только от порта(ов) (интерфейса(ов)), которые созданы для удовлетворения потребностей бизнес-логики и не зависят от конкретного инструмента или адаптера.

Это значить, что **зависимости направяются к центру**, и это **принцип инверсии зависимостей (SOLID: DIP)** на архитектурном уровне.

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



Организация ядра приложения

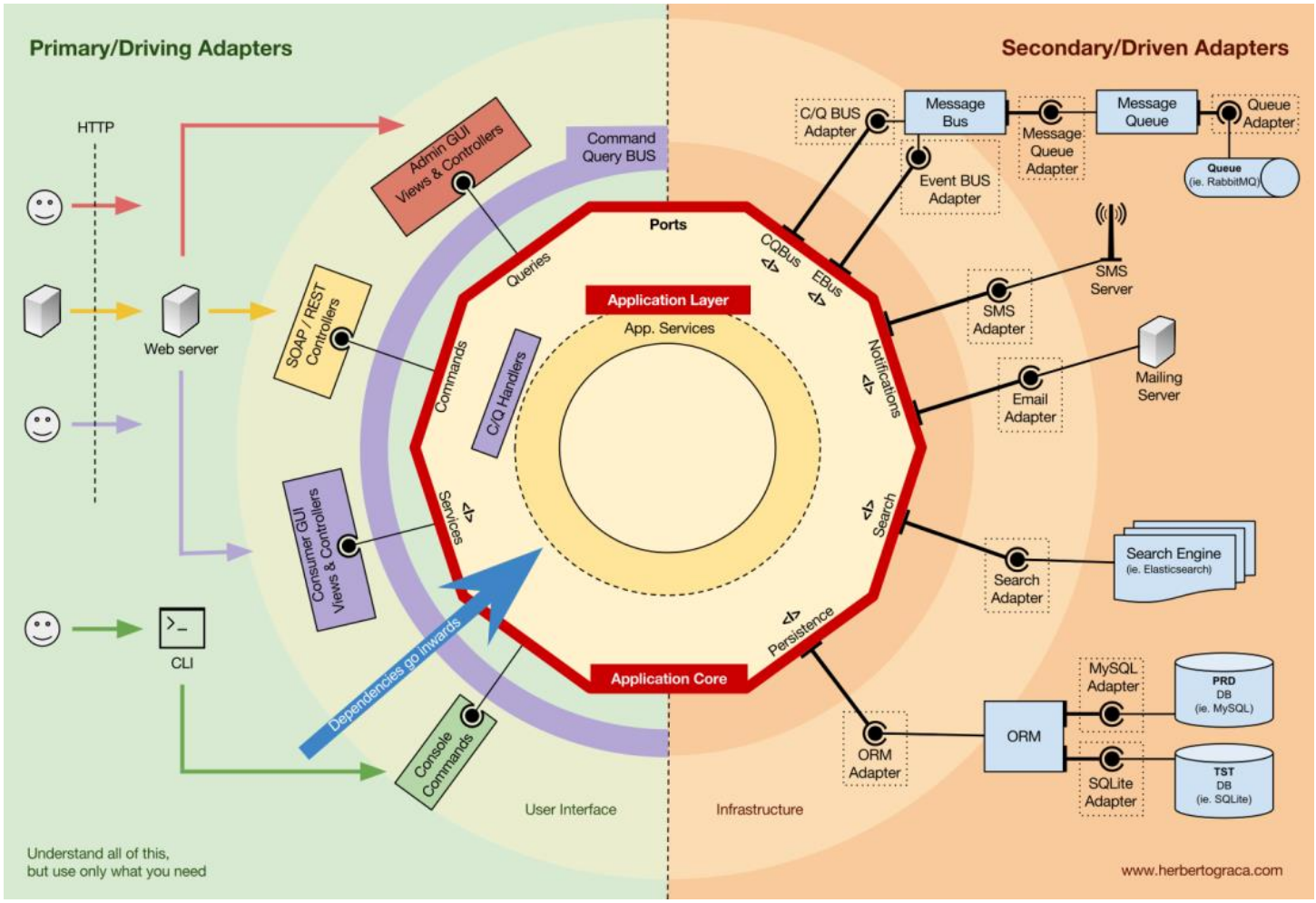
Onion архитектура берет уровни, определенные в DDD (Domain-Driven Design) и встраивает их в Ports & Adapters (Hexagonal) архитектуру. Эти уровни призваны внести опередеенную организацию в бизнес-логику, т.е. во внутреннюю часть (ядро приложения) архитектуры Ports & Adapters (Hexagonal). В Onion зависимости направлены к центру также как в Hexagonal архитектуре.

Уровень приложения (Application Layer)

Варианты использования (use case'ы) – это процессы, которые могут быть запущены в ядре нашего приложения через один или несколько UI. Например, в

CMS у нас может быть UI для обычных пользователей, UI для администратора, CLI UI и Web API. Все эти UI могут запускать варианты использования (use case'ы) специфичные для каждого из них, или совместно использовать общие варианты использования.

Варианты использования (use case'ы) отределяются и размещаются на уровне приложения (Application Layer) – это уровень взятый из DDD и используемый в Onion архитектуре.



Application Layer содержит Application Service'ы (или Use Case'ы) и их интерфейсы (порты), но в тоже время Application Layer содержит ORM интерфейсы (порты), интерфейсы поисковых механизмов, интерфейсы обмена сообщениями и т.д. В случае использования Command Bus и/или Query Bus, этот уровень содержит подходящие обработчики (C/Q Handlers) для Commands и Queries.

Application Service'ы (или Use Case'ы) и/или C/Q Handler'ы содержат логику для запуска варианта использования или какого-то бизнес-процесса. Обычно в их обязанности входит:

1. Используя Repository найти одну или несколько сущностей предметной области (Entity);
2. Попросить эти сущности выполнить определенную в них логику предметной области;
3. Опять использовать Repository для сохранения сущностей (сохранения изменений в их состоянии).

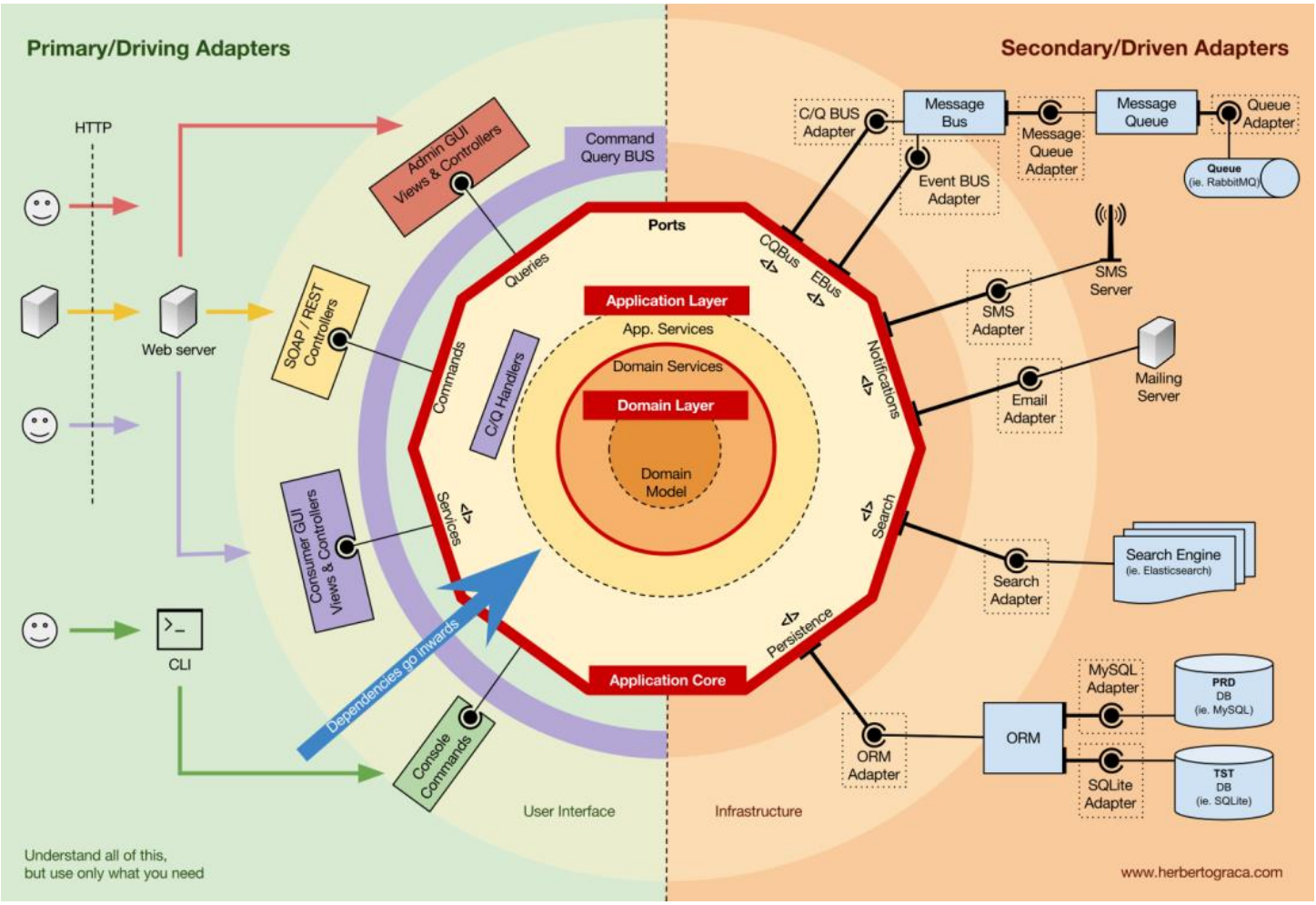
Command Handler'ы могут быть использованы 2 различными способами:

1. Они могут содержать логику для запуска варианта использования;
2. Также они могут быть использованы как промежуточное звено (прослойка), которое принимает Command'у и просто запускает логику, которая определна в уже существующем Application Service'е.

Также на Application Layer'е могут генерироваться Application Event'ы, которые представляют собой определенные побочные эффекты от прохождения варианта использования, например, отправка email'а, вызов стороннего API, отправка push-нотификации или запуск другого варианта использования, который принадлежит другому компоненту приложения.

Уровень предметной области (Domain Layer)

Углубляясь к центру мы приходим к Domain Layer'у. Объекты на этом уровне содержат данные и логику для манипуляции этими данными, которые относятся в предметной области. Объекты на этом уровне независимы от Application Layer'е и ничего не знают о нем.



Сервисы уровня предметной области (Domain Services)

Бывает так, что у нас есть логика предментой области, которая разными сущностями предметной области одинакового или разных типов, и мы ощущаем, что эта логика не пренадлежит этим сущностям, что это не является обязанностью/ответственность этих сущностей.

Таким образом нашей первой реакцией может быть размещение этой логики отдельно от сущностей в Application Service'е (в Application Layer'е). И в результате мы не сможем использовать повторно эту логику в другом варианте использования. Вывод: **логика предметной области не должна расспологаться в Application Layer'е**.

Правильным решением будет создание Domain Service'а, в обязанности которого входит получение набора сущностей и выполнение бизнес-логики над этими сущностями. **Domain Service'ы пренадлежат Domain Layer'у и ничего не знают о классах и объектах Application Layer'а, таких как Application Service'ы, Repository'ии и т.д.** С другой стороны Domain Service'ы могут использовать другие Domain Service'ы, и конечно объекты (сущности, объекты-значения) предметной области.

Модель предметной области (Domain Model)

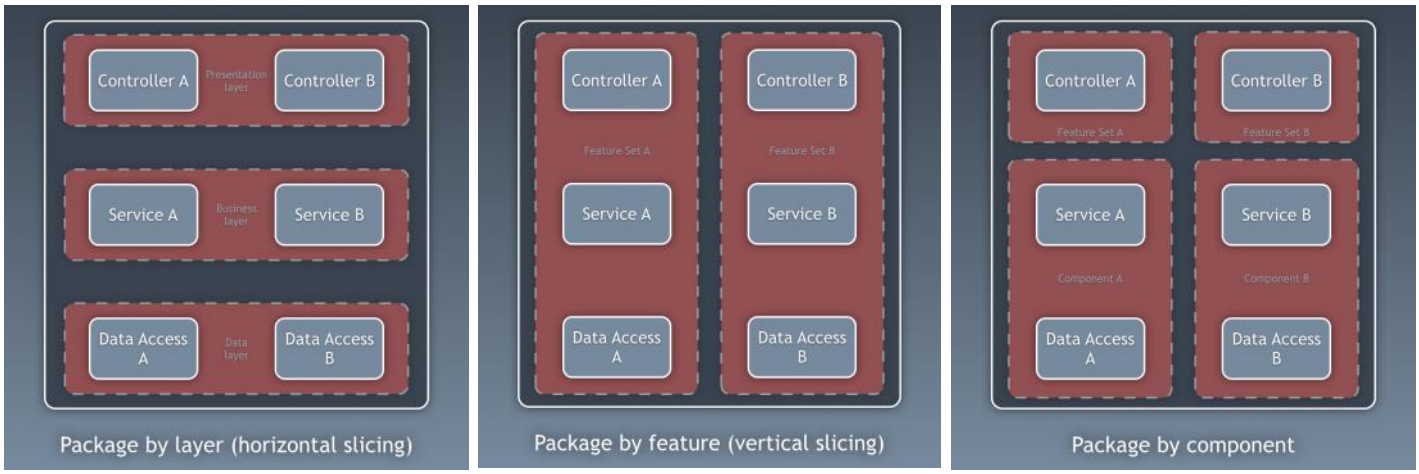
В самом центре, независящая ни от чего и незнающая ни о чем снаружи, находится Domain Model, которая содержит объекты отражающие понятия предметной области. Такими объектами могут быть сущности (Entities), объекты-значения (Value-Objects), перечисления (Enums) и любые другие объекты, используемые в модели предметной области.

Также в Domain Model расспологаются Domain Event'ы. Эти event'ы генерируются, когда изменяется определнный набор данных и эти event's несут эти изменения с собой. Другими словами, когда состояние сущности предметной области изменяется, генерируется event, который несет с собой измененное состояние сущности.

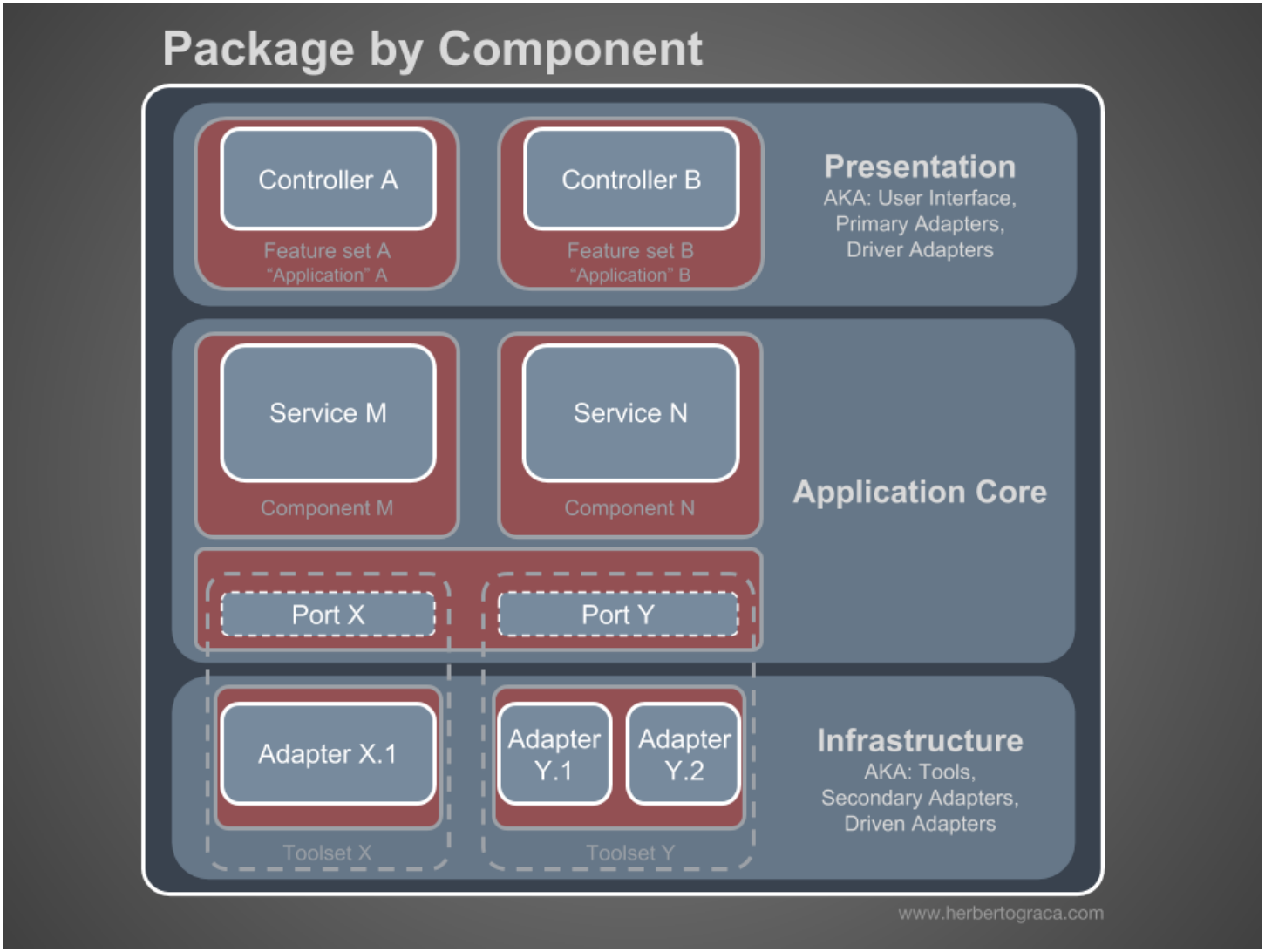
Компоненты

До этого момента мы разделяли код на основе уровней, но это разделение на мелкие части. Разделение кода на крупные части также важно и это разделение производится на основе подобластей предметной области (sub-domains) и ограниченных контекстов (bounded contexts), в соответствии с идеями Роберта Мартина выраженных в "кричащей архитектуре" (screaming architecture). Это чаще относится к разделению "Package by feature" или "Package by component" в

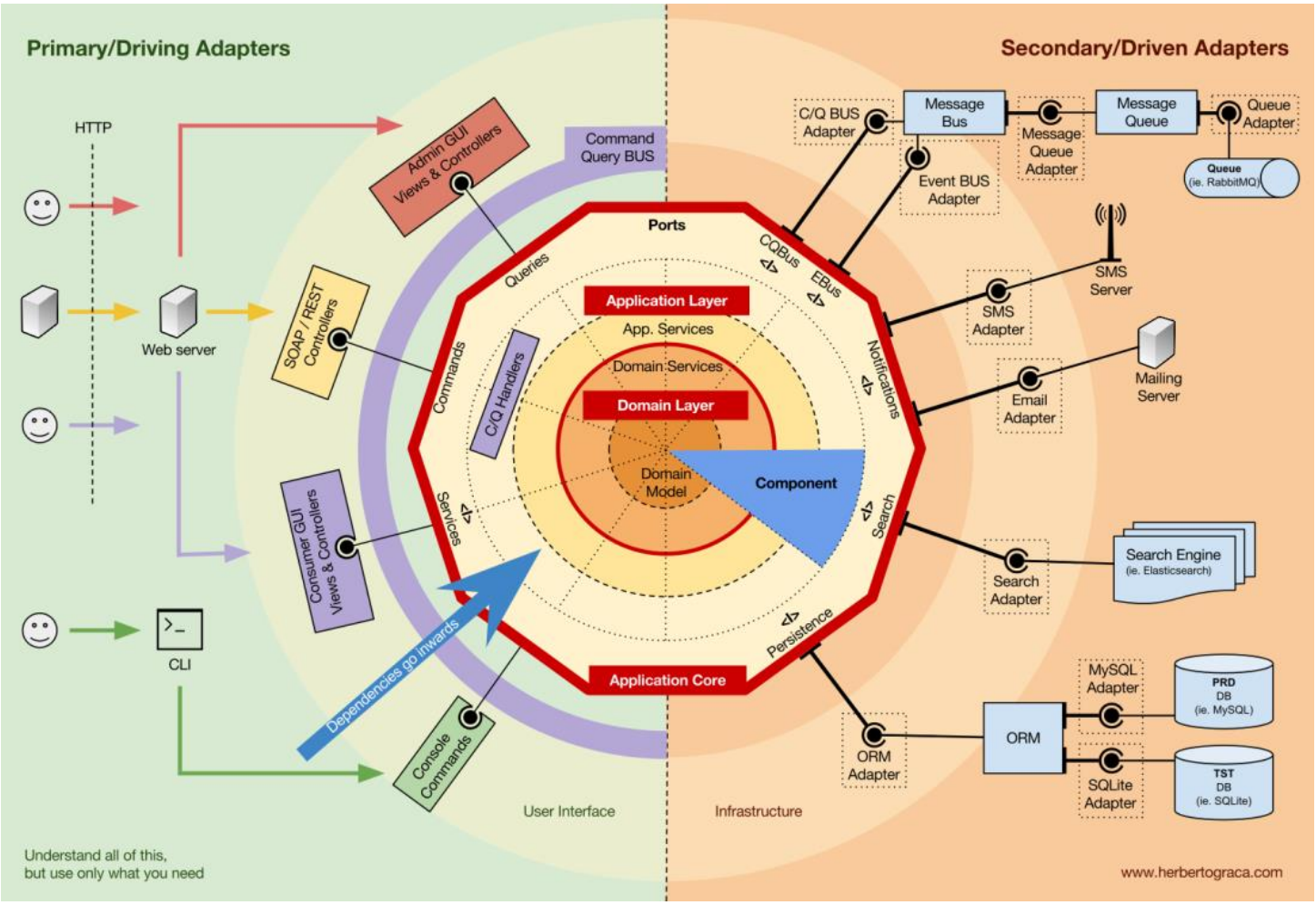
отличии от ”Package by layer“, и это довольно хорошо объяснено Саймоном Брауном в его блоге “Package by component and architecturally-aligned testing“:



Будучи стороннико подхода “Package by component”, я воспользовался соответствующей диаграммой Саймона Брауна и бессовестно ее переделал:



Примерами компонентов могут быть ~~Authentication~~, ~~Authorization~~, Billing, User, Review или Account. **Компоненты всегда относятся к предметной области**. Ограниченные контексты вроде ~~Authentication~~ и/или ~~Authorization~~ должны рассматриваться как сторонние инструменты для каждого из которых у нас есть **port** (интерфейс) в ядре приложения и **adapter** во внешнем слое.



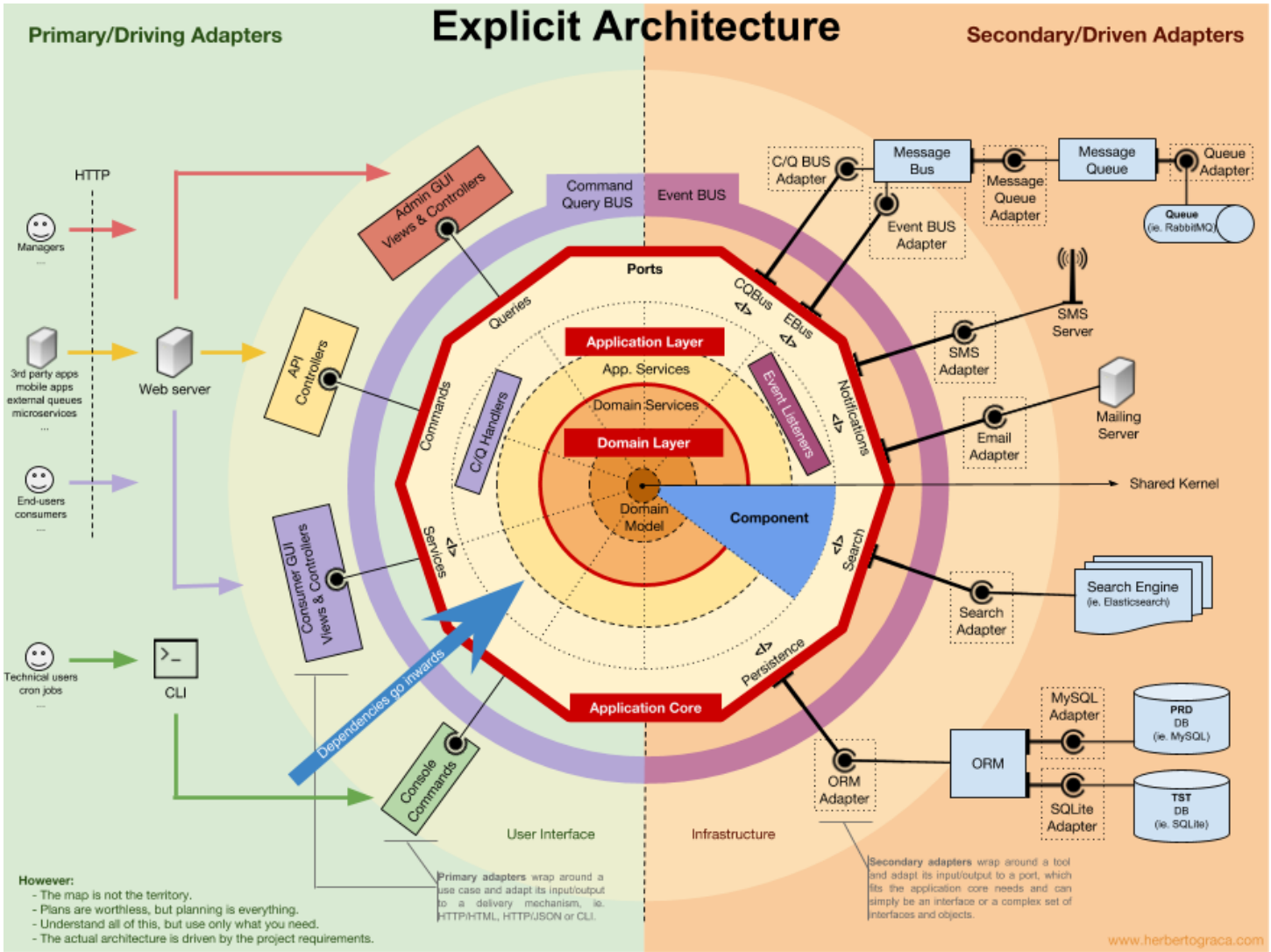
Отделение компонентов друг от друга

Как и мелкие части кода (классы, интерфейсы и т.д.), крупные части кода (компоненты) также получают выгоду от высокой внутренней связности (high cohesion) и низкой внешней зависимости (low coupling).

Для отделения классов друг от друга мы используем **внедрение зависимостей** (Dependency Injection) в качестве альтернативы создания необходимого

экземпляра внутри класса, и **инверсию зависимостей** (Dependency Inversion), делая класс зависимым от абстракции (интерфейс/абстрактный класс), а не от конкретного класса. Т.е. класс, в который внедряется его зависимоть ничего не знает о конкретном классе, который будет использовать.

Аналогично, полностью разделенные компоненты ничего не знают друг о друге. Или другими словами, код одного компонента не имеет ссылок на классы (и даже интерфейсы) любого другого компонента. Это означает, что Dependency Inversion и Dependency Injection не достаточно для отделения компонентов и нам необходимы определенные архитектурные конструкции, которыми могут быть events (события), shared kernel (общее ядро), eventual consistency (согласованность событий) и даже discovery service (служба обнаружения).



Вызов логики другого компонента

Когда один из компонентов, например, **B** должен что-то делать, если что-то произошло в **A**, мы не можем просто сделать вызов метода компонента **B** из **A**, потому как в это случае **A** будет связан с **B**.

Однако мы можем позволить компоненту **A** использовать диспетчер событий для отправки события приложения, которое будет доставлено любому компоненту, прослушивающему его, включая **B**, а прослушиватель событий в компоненте **B** вызовет нужное действие. Это означает, что компонент **A** будет зависеть от диспетчера событий, но будет отделен от **B**.

Тем не менее, если событие (класс события) объявлено в компоненте **A**, то это значит, что **B** знает о существовании **A**, т.е. **B** связан с **A**. Для устранения такой связи мы можем создать библиотеку (общее ядро (Shared Kernel)), которая будет содержать часть функциональности ядра приложения, которая будет видна всем компонентам. Это значит, что компоненты будут зависеть от Shared Kernel, но не будут зависеть друг от друга. Shared Kernel должно содержать события уровня предметной области и уровня приложения, а также любые другие объекты, которые должны быть доступны всем компонентам. Важно, чтобы Shared Kernel было как можно меньше, т.к. любые изменения в нем могут привести к изменениям в компонентах, которые зависят от него. Более того, если наша системы написана на нескольких языках программирования, например, микросервисная система, то общее ядро не должно зависеть от языка, чтобы его могли понимать все компоненты, независимо от языка, на котором они были написаны. Например, Shared Kernel может содержать не классы событий, а описания событий, например, в формате JSON, чтобы все компоненты могли интерпретировать его и, возможно, даже автоматически генерировать свои собственные конкретные реализации.

Этот подход работает как в монолитных приложениях, так и в распределенных приложениях, таких как микросервисные системы. Однако когда действия в

одном компоненте должны приводить к мгновенным изменениями в других компонентах, этот подход не годится, т.к. он осуществляется асинхронно. В такой ситуации компонент **A** должен выполнить HTTP-запрос к компоненту B и чтобы они все также были полностью отделены нам необходим **discovery service** (служба обнаружения). В этой ситуации компонент **A** может спросить у discovery service'а куда ему отправить запрос, или discovery service может работать как proxy, т.е. сам передать запрос нужному компоненту и вернуть его ответ компоненту **A**. В этом случае компоненты будут зависеть от discovery service'а, но не будут зависеть друг от друга.

Получение данных от других компонентов

На мой взгляд компонент не может изменять данные, которыми он не обладает, но это нормально, когда компонент запрашивает и использует данные других компонентов.

Хранилище данных, доступное всем компонентам

Когда компонент должен использовать данные, которые принадлежат другому компоненту, например, Billing компонент должен использовать имя клиента, которое принадлежит компоненту Accounts. В таком случае Billing компонент должен содержать объект запроса (query-object), который запрашивает из хранилища данных это имя клиента. Это означает, что Billing компонент может знать о любом наборе данных, но он должен использовать данные, которыми он не "владеет", только для чтения.

Отдельное хранилище данных для каждого компонента

В данном случае мы имеем больше сложности на уровне хранения данных. Наличие компонентов с собственным хранилищем данных означает, что каждое хранилище данных содержит:

- Набор данных, которыми владеет компонент и только этот компонент может их изменить, что делает данное хранилище данных единственным источником истины по поводу корректности этих данных;
- Набор данных, который является копией данных других компонентов. Данный комопнент (владелец хранилища) не может изменить этот набор данных, но он необходим компоненту для функционирования. Также этот набор данных необходимо обновлять всякий раз, когда он изменяется компонентом владельцем в своем хранилище.

Каждый компонент создает локальную копию данных других компонентов, которые необходимы ему для функционирования. При изменении данных компонентом-владелецем, компонент-владелец инициирует событие предметной области (domain event), которое несет изменения данных. Компоненты, содержащие копию этих данных, будут прослушивать событие предметной области и соответствующим образом обновлять свою локальную копию данных.

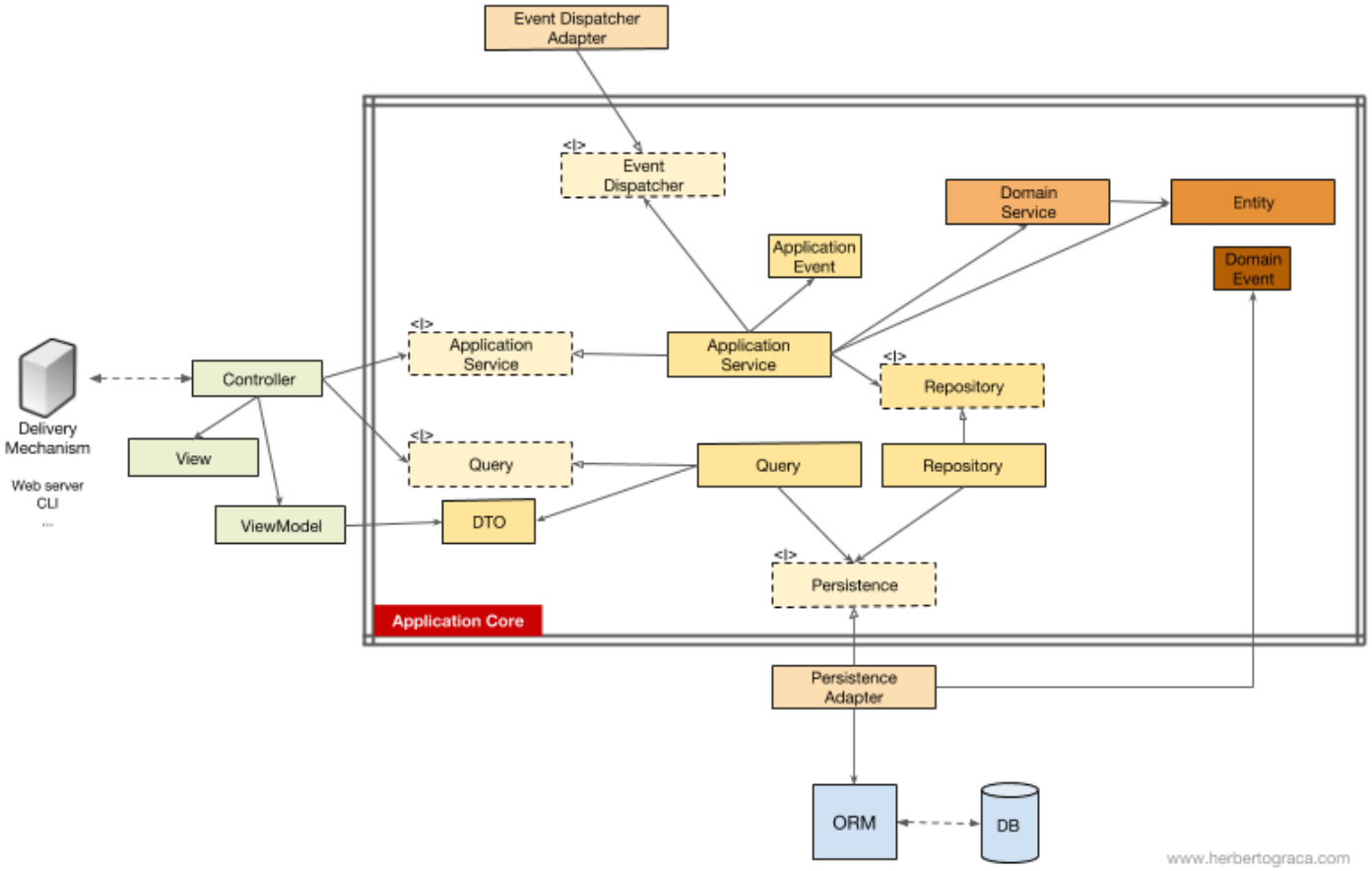
Поток управления

Как было упомянуто выше, поток управления идет, от пользователя >>> в ядро приложения >>> к инструментам инфраструктуры >>> обратно в ядро приложения >>> и обратно к пользователю. Но как именно классы взаимодействуют друг с другом? Какие классы зависят от каких? Как мы объеденяем классы вместе?

Опираясь на статью Роберта Мартина о Clean Architecture, я попытаюсь объяснить поток управления с использование UML-диаграмм...

Без использования Command/Query Bus

В случае, если мы не используем C/Q Bus наши Controller'ы могут зависеть как от Application Service'ов, так и от Query Object'ов.



На приведенной выше диаграмме мы используем интерфейс для Application Service'а, хотя можно утверждать, что на самом деле он и не нужен, так как Application Service является частью Application Core, и мы не хотим менять реализацию Application Service'а на другую.

Query Object будет содержать оптимизированный запрос, который будет просто возвращать некоторые необработанные данные, которые будут показаны пользователю. Эти данные будут возвращены в виде DTO, который будет передан во ViewModel. ViewModel, в свою очередь, может содержать какую-то логику

представления этих данных, и она будет использоваться для заполнения представления данных пользователю (View).

Application Service, напротив, будет содеражать логику вариантов использования (Use Case'ов) и эта логика активизируется, когда мы хотим что-то сделать в системе, как противоположность простому отображению данных с использованием Query Object. Application Service зависит от Repository'ев, возвращающих сущности предметной области (Entities), которые содержат логику предметной области, которая должна быть выполнена. Application Service так же может зависеть от Domain Service'ов, которые координируют процессы предметной области, манипулируя несколькими сущностями предметной области, но так бывает редко.

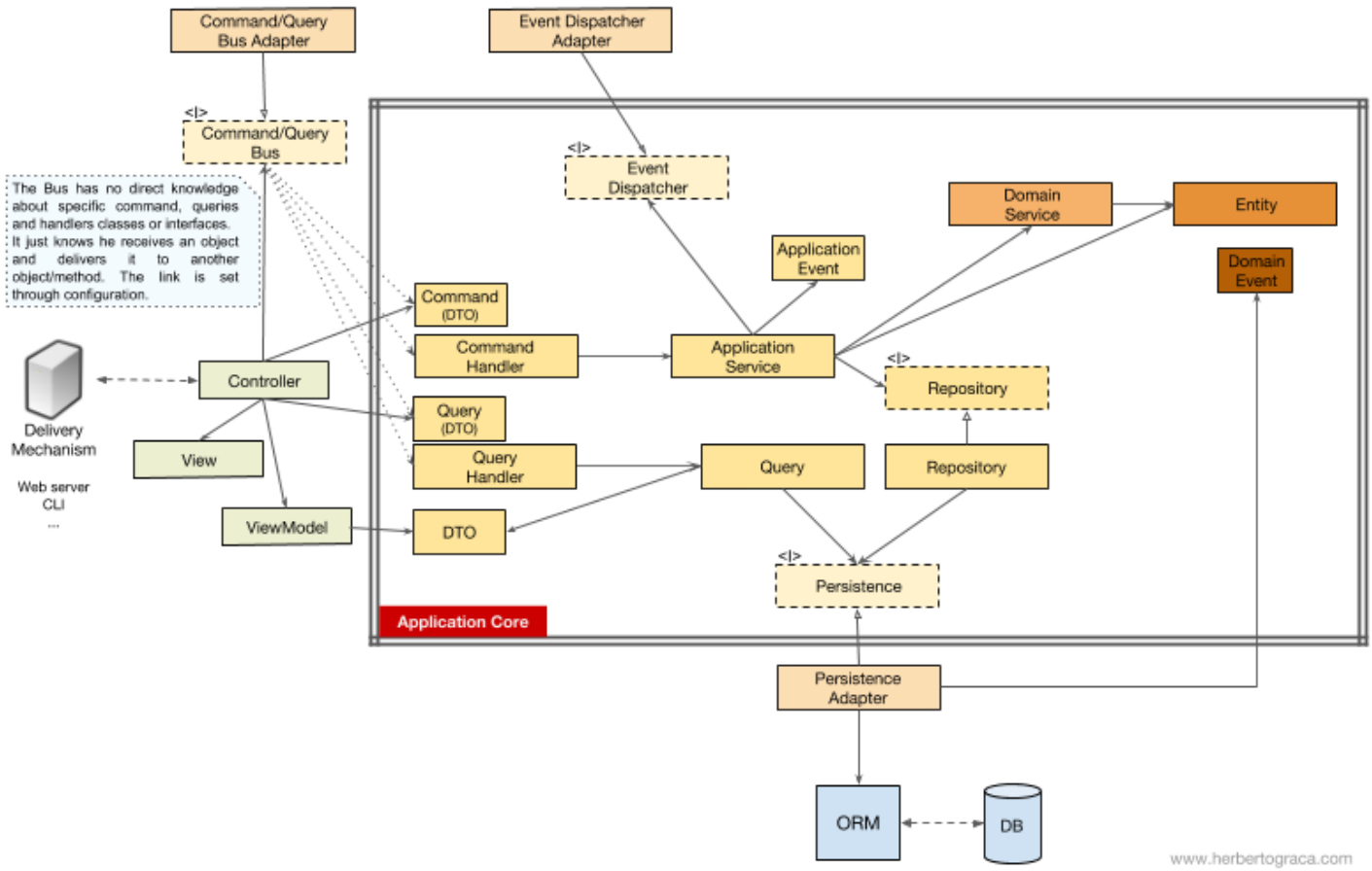
После выполнения Use Case'а Application Service'у может понадобиться оповестить всю систему о том, что Use Case был выполнен. В таком случае Application Service будет так же зависеть от Event Dispatcher'а, чтобы инициировать необходимое событие.

Интересно заметить, что мы имеем интерфейс как для механизма отображения данных/объектов (Persistence Mechanism), так и для Repository'ев. Хотя это может показаться излишним, но они служат различным целям:

- Интерфейс Persistence Mechanism'а – это абстракция над ORM, которая позволяет менять ORM-реализацию (в Java это могут быть JPA реализации: Hibernate, Eclipse Link и т.п.) без изменений в Application Core'е;
- Интерфейс Repository'я – это абстракция над Persistence Mechanism'омо. Допустим мы хотим мигрировать с MySQL на PostgreSQL. Persistence Mechanism (ORM реализация) может оставаться остается тем же. Однако при данном переходе язык запросов будет отличаться, поэтому нам необходимо создать новую реализацию интерфеса Repository'я, которая будет использовать тот же Persistence Mechanism, но использовать синтаксис запросов PostgreSQL вместо MySQL.

С использованием Command/Query Bus

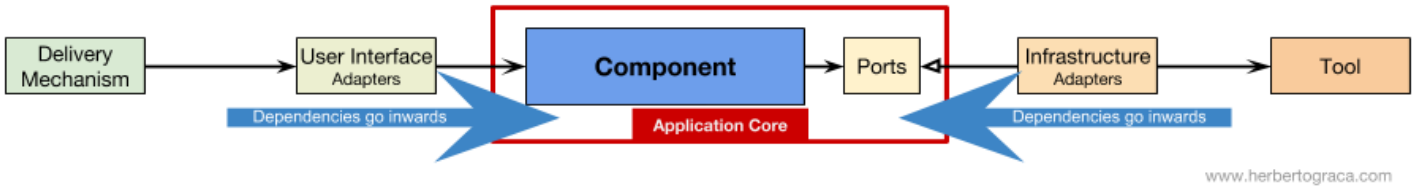
В случае, если мы используем C/Q Bus диаграмма остается почти такой же, с тем отличием, что Controller будет зависеть от Bus'а и от Command или Query. Т.е. Controller создает объект Command или Query и передает его Bus'у, который в свою очередь находит подходящий Handler для обработки этой Command или Query.



На диаграмме Handler использует Application Service. Однако это не всегда необходимо, на самом деле в большинстве случаев Handler может содержать всю логику варианта использования (use case'а). Извлекать логику use case'а из Handler'а в отдельный Application Service необходимо только, если нам нужно повторно использовать ту же логику use case'а в нескольких Handler'ах.

Возможно, вы заметили, что нет никакой зависимости между Bus'ом, Command/Query и Handler'ом. Это потому, что они должны, по сути, не знать друг о друге, чтобы обеспечить хорошее отделение. Из кофигурации Bus должен получать информацию о том, какой Handler должен обработать какую Command/Query.

Как вы видите, на обоих диаграммах приведенных выше все стрелки (зависимости), которые пересекают границу Application Core'а, указывают внутрь. Как объяснялось ранее, это фундаментальное правило таких архитектур как Ports & Adapters (Hexagonal) Architecture, Onion Architecture и Clean Architecture.



Заключение

Главной целью, как всегда, является получение системы, модули и компоненты которой имеют высокую внутреннюю связность и низкую внешнюю зависимость, т.е. системы, в которую можно легко, быстро и безопасно вносить изменения.

Планы бесполезны, но планирование – это все. (Эйзенхауэр)

Эта статья является концептуальной картой. Знание и понимание понятий изложенных в ней поможет нам планировать качественную архитектуру и создавать качесвенные приложения.

Однако:

Карта не является территорией. (Коржибски)

Это означает, что все вышеизложенное – просто рекомендации! Приложение – это территория, реальность, конкретный случай использования, где нам нужно применить наши знания, и это то, что будет определять, как будет выглядеть архитектура!

Мы должны понимать все вышеизложенное, но мы также всегда должны думать и понимать, что именно нужно нашему приложению, как далеко мы должны зайти ради повышения внутренней связности и снижения внешней зависимости между частями системы. Это решение может зависеть от множества факторов: функциональные требования проекта, временные рамки для создания приложения, срок службы приложения, опыт команды разработчиков и т.д.