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

Перевод статьи: [DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together](https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/) ([Herberto Graca](https://herbertograca.com/))

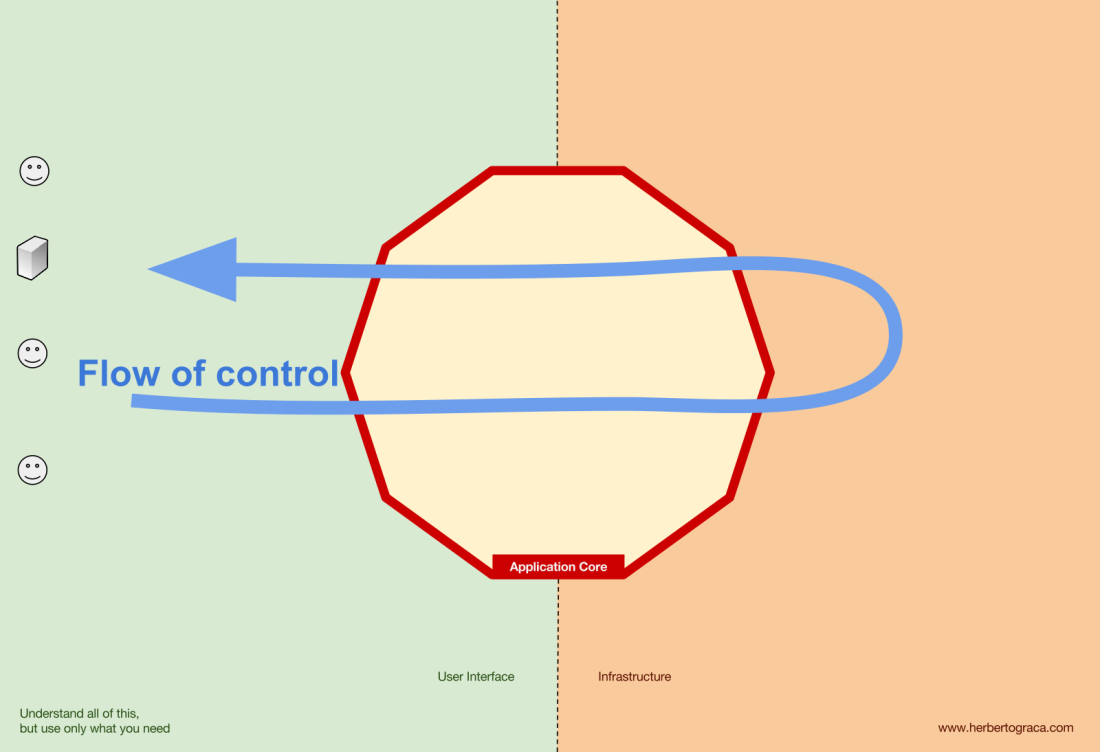
В этой статье я попытался объединить лучшие архитектурные практики, принципы и шаблоны вместе. То что в результате получилось я назвал **явной архитектурой** (Explicit Architecture).

**Фундаментальные блоки системы**

Я начну с упоминания [EBI](https://herbertograca.com/2017/08/24/ebi-architecture/) и [Ports & Adapters (Hexagonal)](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) 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)](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) 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.

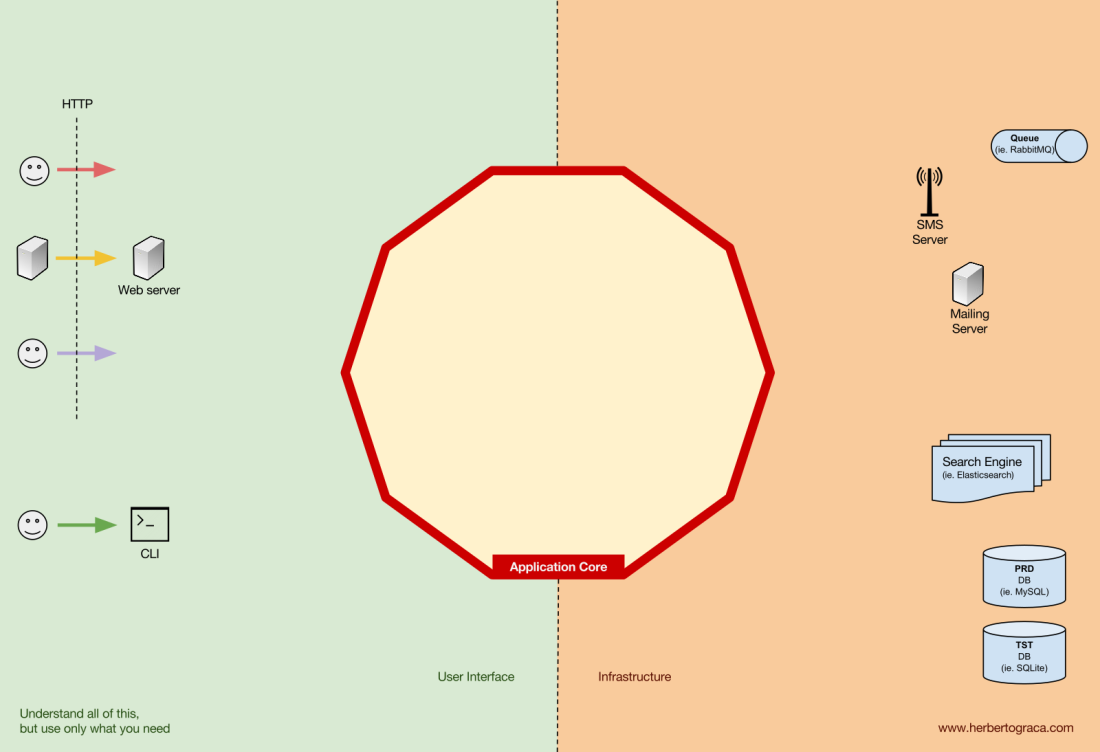


Ядро приложения (Application Core) – это то, о чем мы действительно должны заботиться. Оно может быть использовано разными UI (Web-интерфейс, мобильное приложение, CLI, API, …).

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)](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) 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**).

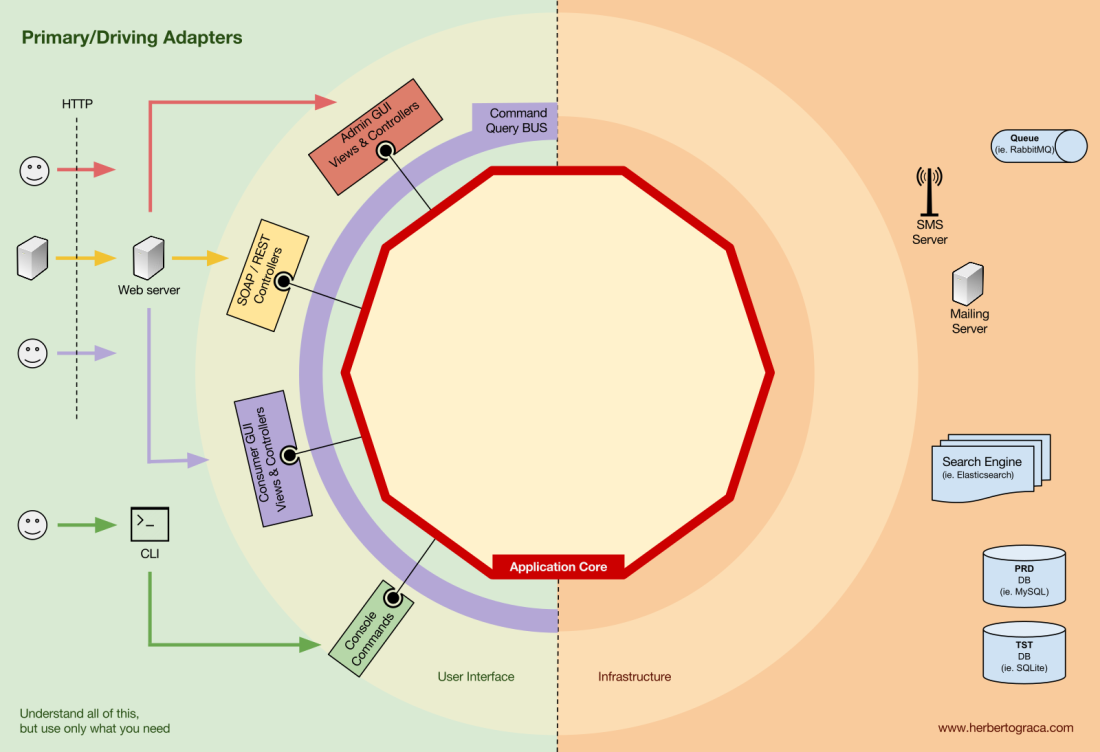
**Порты**

Однако эти 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 бизнес-логики, 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).



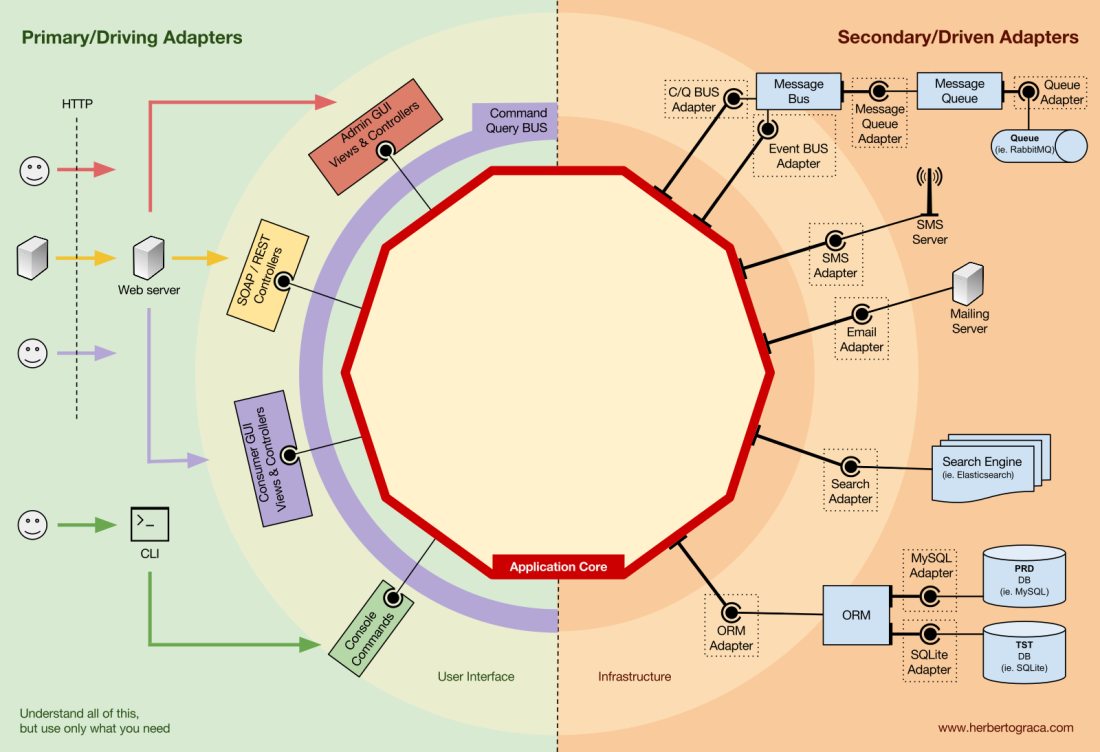
Другими словами, Управляющие Адаптеры – это Controller’ы или Console Command’ы, в которые внедряется (через конструктор или любым другим способом) объект, который реализует интерфейс (порт), который необходим Controller’у или Console Command’е для взаимодействия с ядром приложения.

Более конкретным примером может служить интерфейс сервиса (Service Interface) или интерфейс хранилища (Repository Interface), и далее конкретная реализаци Service’a, Repository или Query далее внедряется и используется, например, в Controller’е.

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

**Вторичные или Управляемые Адаптеры**

В отличии от Управляющих Адаптеров, которые содержат порт, **Управляемые Адаптеры реализуют 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 мотодом 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.

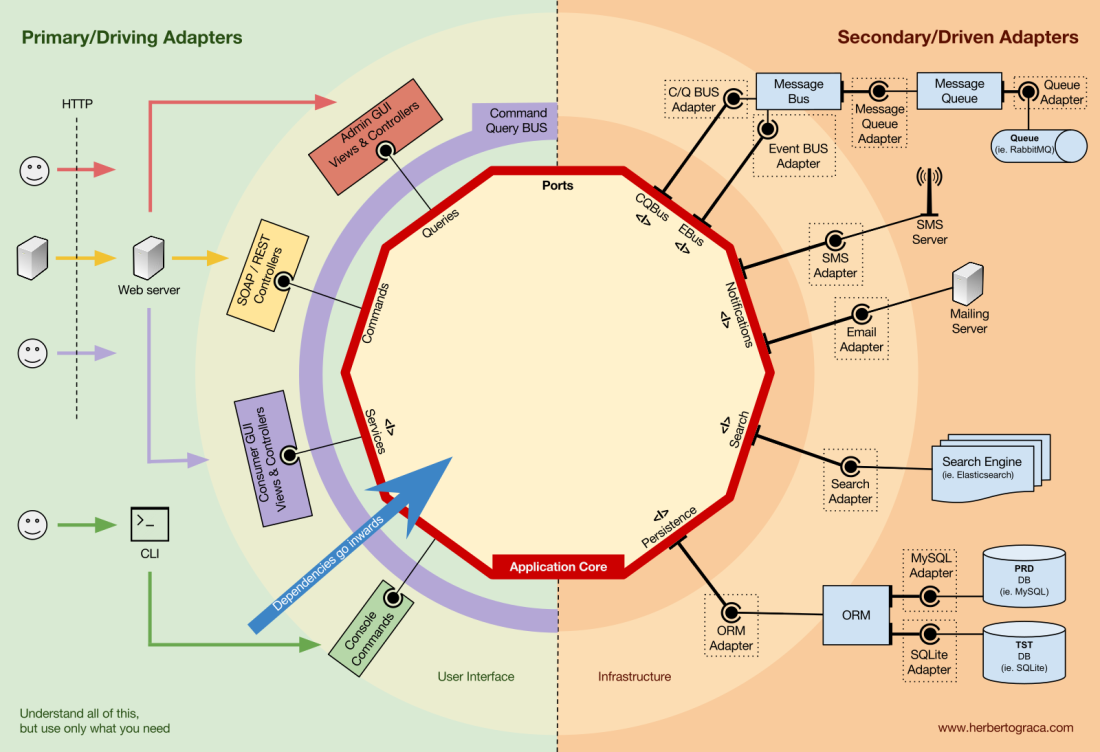
Если в какой-то момент мы решим сменить поставщика БД, например, на PostgreSQL или MongoDB, то нам будет необходимо только создать адаптер, который реализует наш интфейс (порт) и спецефичен для, например, PostgreSQL, и далее внедрить новый адаптер вместо старого.

**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)), который создан для удовлетворения потребностей бизнес-логики и не зависит 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**.



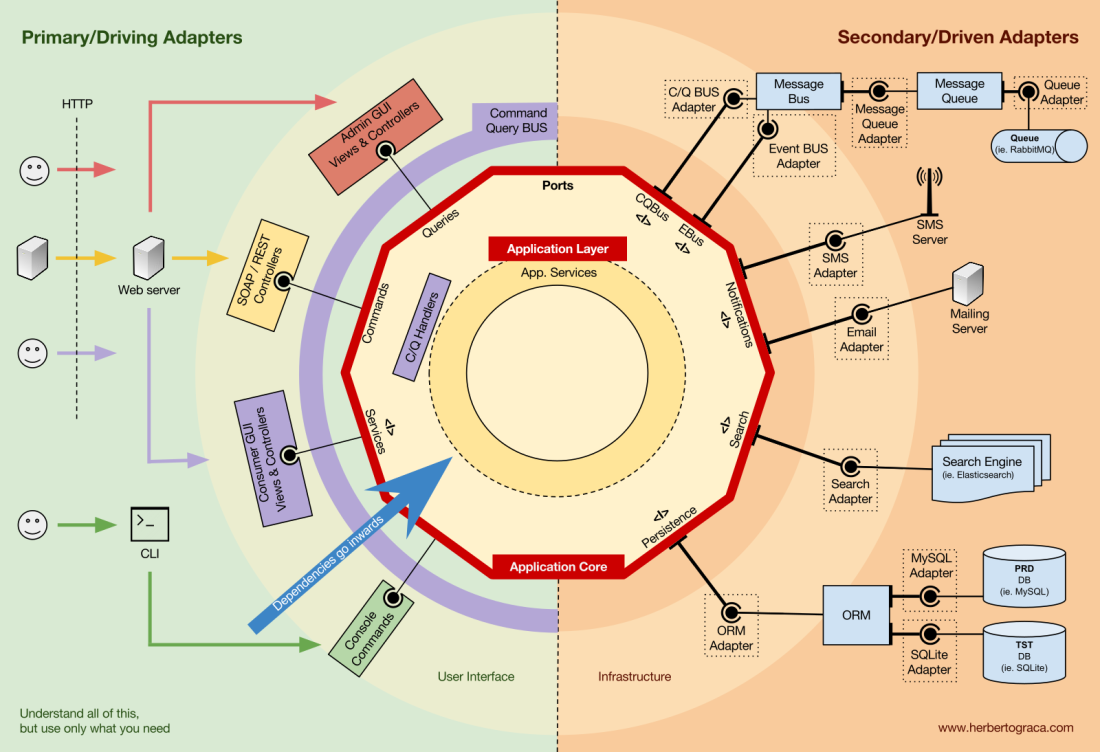
**Организация ядра приложения**

[Onion](https://herbertograca.com/2017/09/21/onion-architecture/) архитектура берет уровни, определенные в DDD (Domain-Driven Design) и встраивает их в [Ports & Adapters (Hexagonal)](https://herbertograca.com/2017/09/14/ports-adapters-architecture/) архитектуру. Эти уровни призваны внести опеределенную организацию в бизнес-логику, т.е. во внутреннюю часть (ядро приложения) архитектуры [Ports & Adapters (Hexagonal)](https://herbertograca.com/2017/09/14/ports-adapters-architecture/). В [Onion](https://herbertograca.com/2017/09/21/onion-architecture/) 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](https://herbertograca.com/2017/09/21/onion-architecture/) архитектуре.



Application Layer содержит Application Service’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 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;
2. Попросить эти сущности выполнить определеннуй в них логику предметной области;
3. Опять использовать Repository для сохранения сущностей (сохранения изменений в их состоянии).

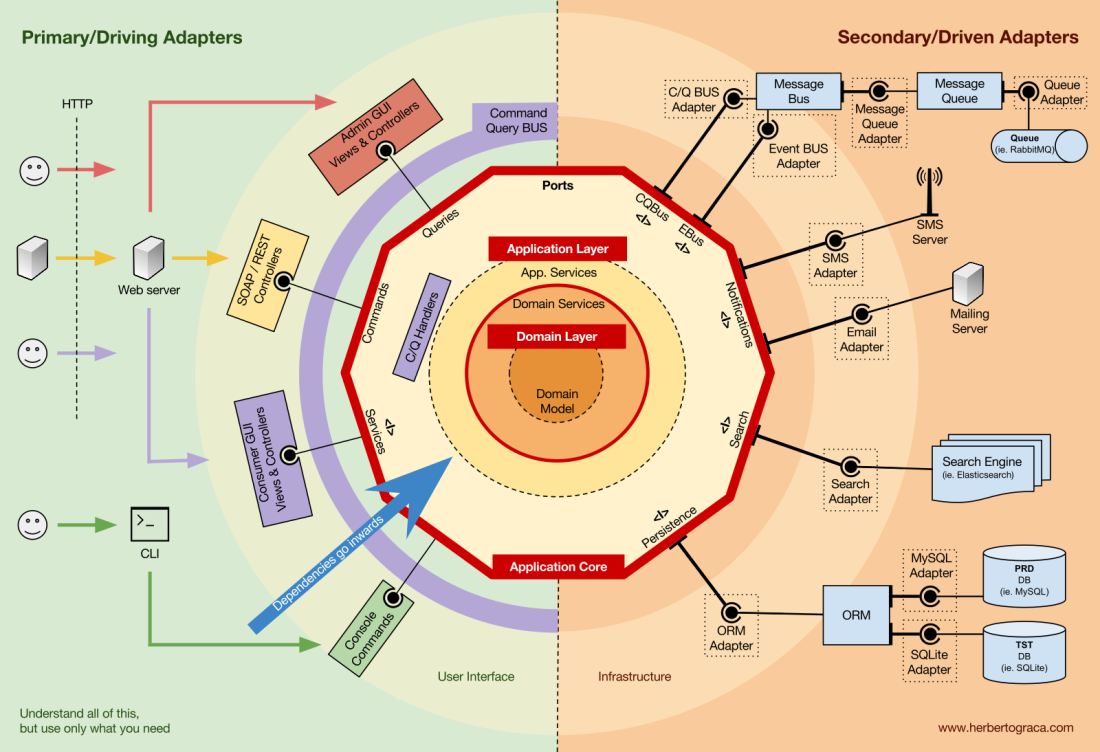
Command Handler’ы могут быть использованы 2 различными способами:

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

Также на Application Layer’e могут генерироваться Application Event’ы, которые представляют собой определенные побочные эффекты от прохождения варианта использования, например, отправка email’a, вызов строннего 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’e 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’e). И в результате мы не сможем использовать повторно эту логику в другом варианте использования. Вывод: **логика предметной области не должна расспологаться в Application Layer’e**.

Правильным решением будет создание Domain Service’а, в обязанности которого входит получение набора сущностей и выполнение бизнес-логткт 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.

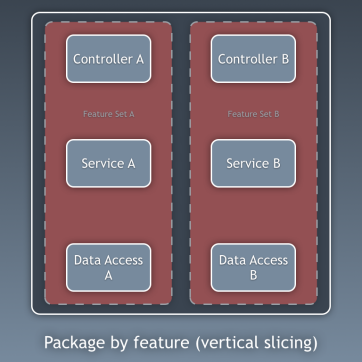
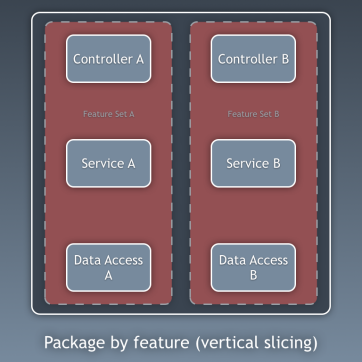
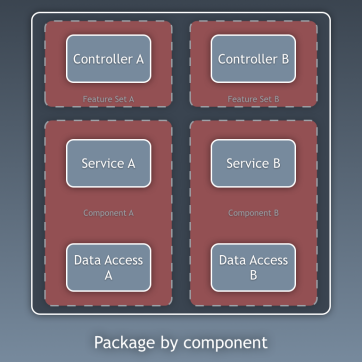
**Модель предметной области (Domain Model)**

В самом центре, независящая ни от чего и знающая 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.

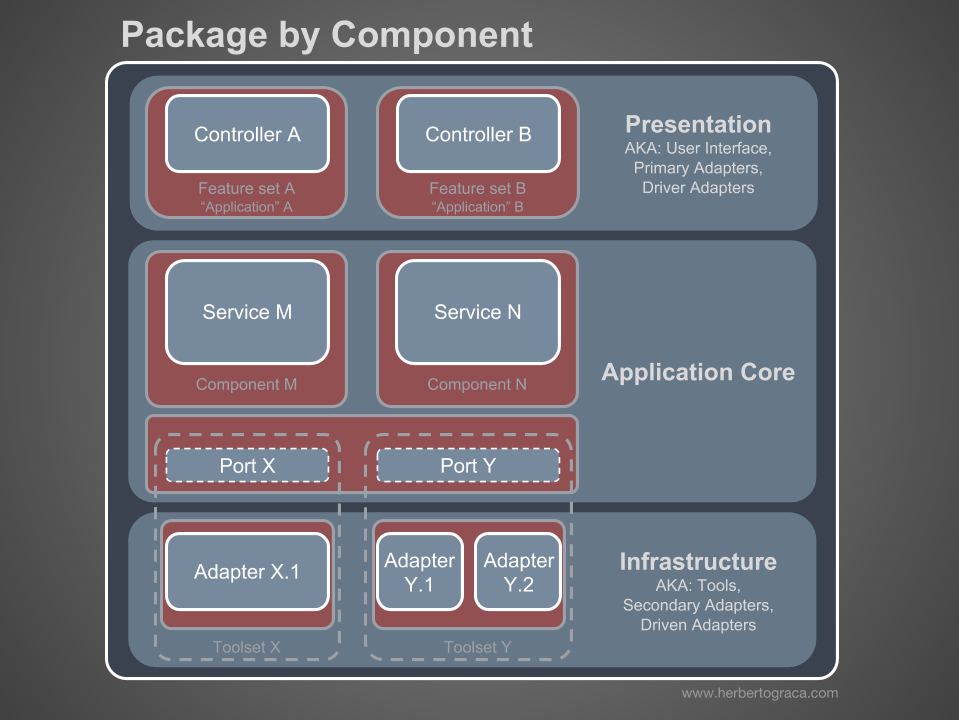
Также в Domain Model расспологаются Domain Event’ы. Эти event’ы генерируются, когда изменяется определнный набор данных и эти event’s несут эти изменения с собой. Другими словами, когда состояние сущности предметной области изменяется, event генерируется и несет вместе с собой это измененное состояние.

**Компоненты**

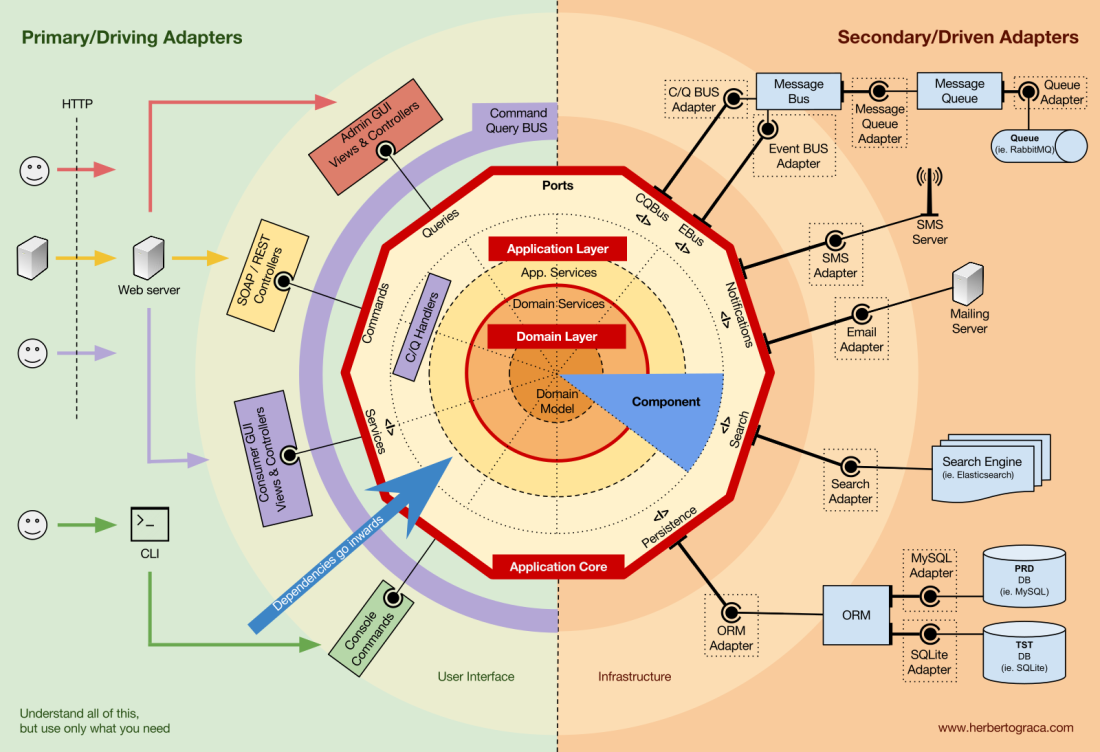
До этого момента мы разделяли код на основе уровней, но это разделение на мелкие части. Разделение кода на крупные части также важно и это разделение производится на основе подобластей предметной области (sub-domains) и ограниченных контекстов ([bounded contexts](http://ddd.fed.wiki.org/view/welcome-visitors/view/domain-driven-design/view/bounded-context)), в соответствии с идеями Роберта Мартина выраженных в “кричащей архитектуре” ([screaming architecture](https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html)). 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](http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html)“:

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



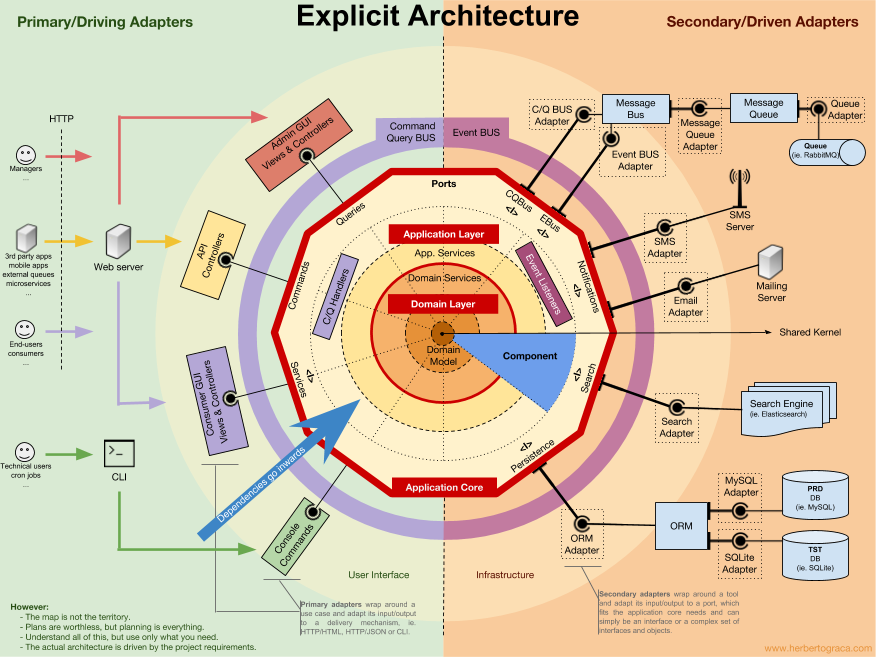
Примерами компонентов могут быть ~~Authentication~~, ~~Authorization~~, Billing, User, Review или Account. **Компоненты всегда относятся к предметной области**. Ограниченные контексты вроде ~~Authentication~~ и/или ~~Authorization~~ должны рассматриваться как сторонние инструменты для которых у нас есть **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). [](https://docs.google.com/drawings/d/1E_hx5B4czRVFVhGJbrbPDlb_JFxJC8fYB86OMzZuAhg/edit)

**Вызов логики другого компонента**

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

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