# DDD, Hexagonal, Onion, Clean, CQRS 组合

<https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/>

这篇文章是软件架构编年史的一部分，这是一系列关于软件架构的文章。在它们中，我写了关于软件架构的知识，我如何看待它，以及我如何使用这些知识。如果你读过本系列的前几篇文章，你会发现这篇文章的内容更有意义。

**系统的基本模块**

我首先回顾一下EBI和端口与适配器体系结构。两者都明确区分了哪些代码是应用程序的内部代码，哪些代码是外部代码，以及哪些代码用于连接内部代码和外部代码。

此外，端口和适配器体系结构明确地标识了系统中的三个基本代码块:

1 user interface： 是什么让运行用户界面成为可能，不管是什么类型的用户界面;

2 business logic, or application core： 系统业务逻辑，或应用程序核心，它被用户界面用来实际使事情发生;

3 Infrastructure ：基础架构代码，将我们的应用核心与数据库、搜索引擎或第三方api等工具连接起来。

应用程序核心是我们真正应该关心的。是代码让我们的代码做它应该做的事情，它是我们的应用程序。它可能会使用几个用户界面(渐进web应用程序，移动，CLI, API，等等)，但实际执行工作的代码是相同的，并且位于应用程序的核心，它不应该真正在乎是什么UI触发它。

正如您可以想象的那样，典型的应用程序流从用户界面中的代码，经过应用程序核心到基础设施代码，再返回到应用程序核心，最后向用户界面交付响应。

**工具**

与系统中最重要的代码(应用程序核心)相去不远的是，我们有应用程序使用的工具，例如数据库引擎、搜索引擎、Web服务器或CLI控制台(尽管最后两个也是交付机制)。

虽然将CLI控制台与数据库引擎放在同一个“bucket”中可能会有些奇怪，而且尽管它们有不同类型的用途，但它们实际上是应用程序使用的工具。关键的区别在于，虽然CLI控制台和web服务器是用来告诉我们的应用程序做一些事情的，但是数据库引擎是由我们的应用程序告诉的。这是一个非常相关的区别，因为它对我们如何构建连接这些工具和应用程序核心的代码有很强的暗示作用。

**将工具和交付机制连接到应用程序核心**

将工具连接到应用程序核心的代码单元称为适配器(端口和适配器体系结构)。适配器是那些有效地实现代码的适配器，这些代码将允许业务逻辑与特定的工具通信，反之亦然。

告诉应用程序执行某些操作的适配器称为主适配器或驱动适配器，而应用程序告知执行某些操作的适配器称为辅助适配器或驱动适配器。

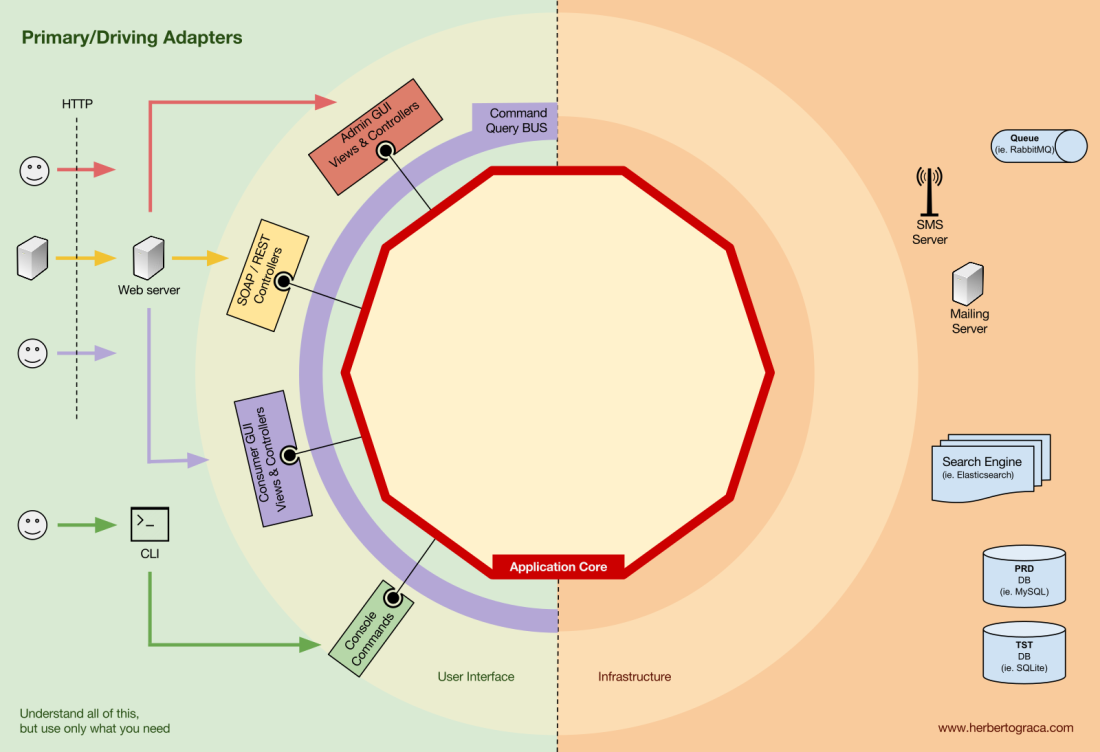
**端口.**

但是，这些适配器不是随机创建的。创建它们是为了适合应用程序核心(端口)的特定入口点。一个端口只不过是一个关于工具如何使用应用程序核心，或者应用程序核心如何使用它的规范。在大多数语言及其最简单的形式中，这个规范(端口)将是一个接口，但它实际上可能由几个接口和dto组成。

需要注意的是，端口(接口)属于业务逻辑内部，而适配器属于业务逻辑外部。为了让这个模式正常工作，创建端口以满足应用程序的核心需求，而不是简单地模仿工具api，这一点至关重要。

**主适配器或驱动适配器**

主适配器或驱动适配器围绕一个端口，并使用它来告诉应用程序核心要做什么。它们将来自交付机制的任何内容转换为应用程序核心中的方法调用。



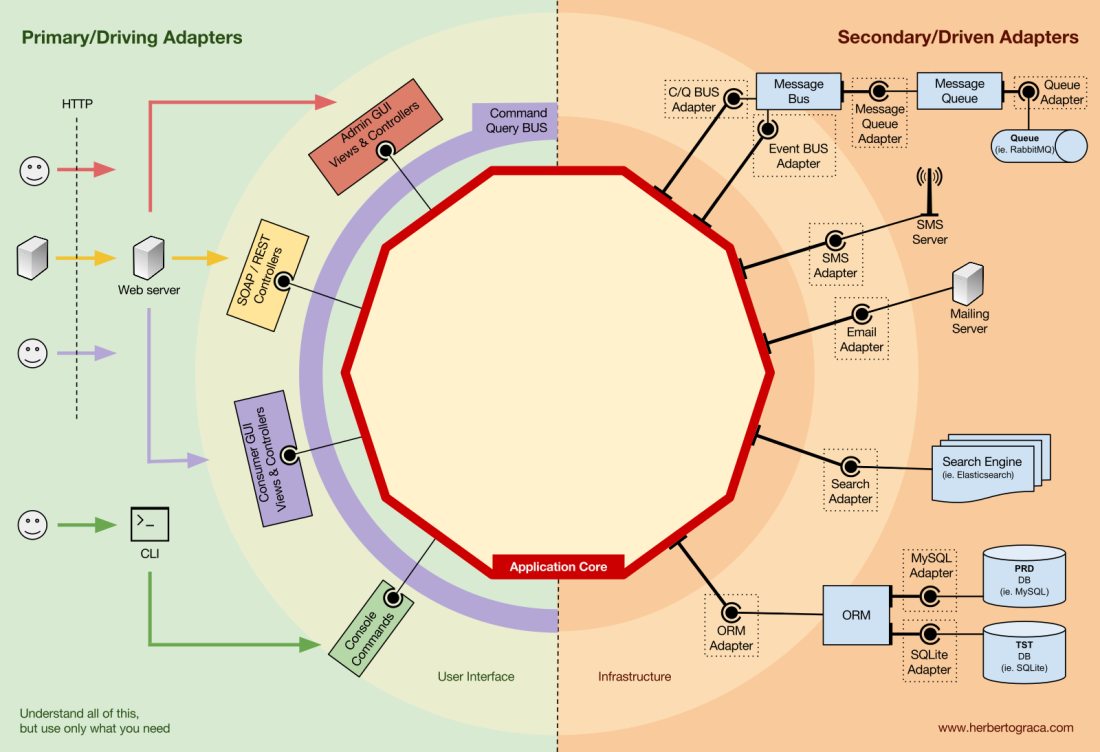
换句话说，我们的驱动适配器是控制器或控制台命令，它们被注入到它们的构造函数中，其中的一些对象实现了控制器或控制台命令所需的接口(端口)。

在更具体的例子中，端口可以是控制器需要的服务接口或存储库接口。然后注入服务、存储库或查询的具体实现，并在控制器中使用。

或者，端口可以是命令总线或查询总线接口。在这种情况下，将注入命令总线或查询总线的具体实现

**辅助或驱动适配器**

驱动适配器与封装端口的驱动适配器不同，驱动适配器实现一个端口、一个接口，然后在需要端口的地方(类型暗示)注入到应用程序核心中。



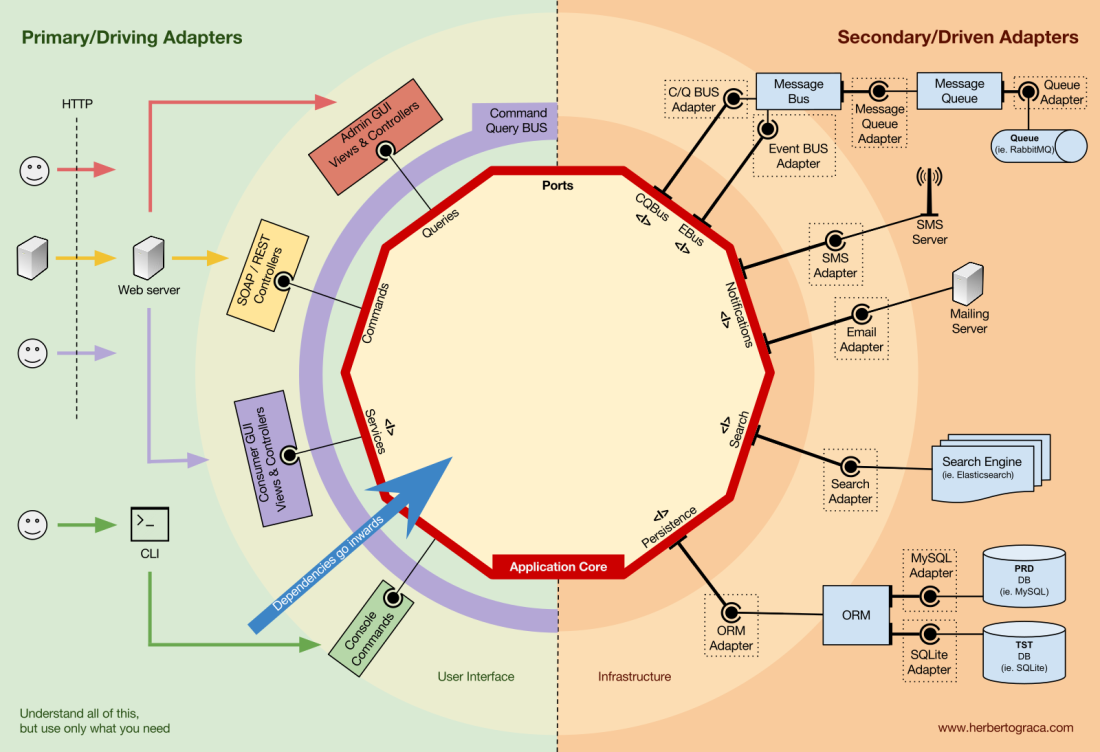
例如，假设我们有一个需要持久化数据的简单应用程序。所以我们创建一个持久性接口,满足其需要,用一个方法来保存数组的数据和方法来删除表中的一行的ID。从那时起,无论应用程序需要保存或删除数据,我们需要在其构造函数实现持久化的对象我们定义的接口。

现在我们创建一个特定于MySQL的适配器，它将实现该接口。它将具有保存数组和删除表中的一行的方法，我们将在需要persistence接口的任何地方注入它。

如果在某个时候我们决定改变数据库供应商，比如PostgreSQL或MongoDB，我们只需要创建一个新的适配器来实现持久化接口，并且是特定于PostgreSQL的，然后注入新的适配器而不是旧的。

**控制反转**

关于此模式需要注意的一个特征是，适配器依赖于特定的工具和特定的端口(通过实现接口)。但是我们的业务逻辑只依赖于端口(接口)，它是为满足业务逻辑需求而设计的，因此它不依赖于特定的适配器或工具。



这意味着依赖的方向是向中心的，这是架构级的控制原则的反转。

不过，同样重要的是，创建端口是为了满足应用程序的核心需求，而不是简单地模仿工具api。

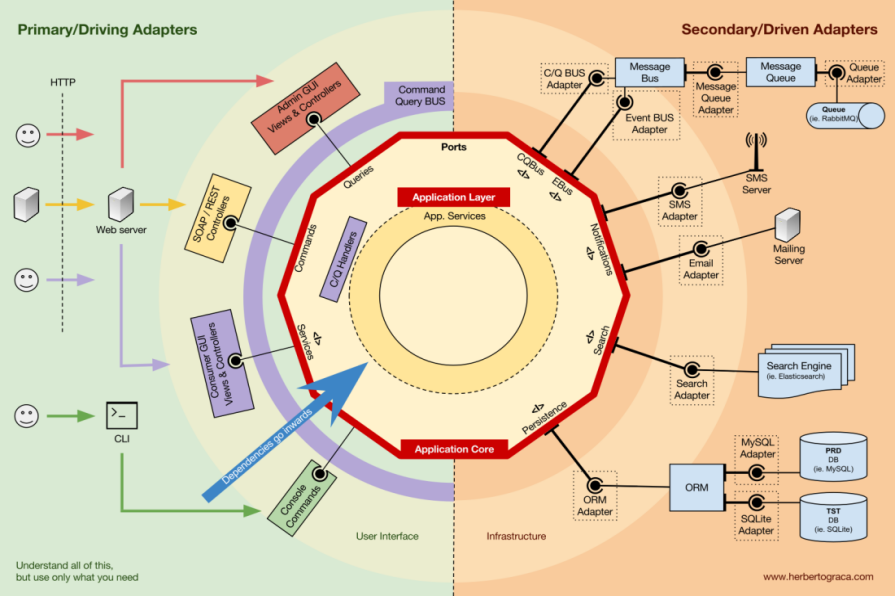
**应用程序的核心组织**

Onion架构拾取DDD层并将它们合并到端口和适配器架构中。这些层旨在为业务逻辑、端口和适配器的内部“六边形”带来一些组织，就像在端口和适配器中一样，依赖关系的方向是向中心的

**应用程序层**

用例是可以由应用程序中的一个或多个用户界面在应用程序核心中触发的流程。例如，在CMS中，我们可以拥有普通用户使用的实际应用程序UI、CMS管理员使用的另一个独立UI、另一个CLI UI和一个web API。这些ui(应用程序)可以触发特定于其中一个ui或由多个ui重用的用例。

用例在应用层中定义，这是DDD提供的第一层，由Onion体系结构使用。



这一层包含应用程序服务(及其接口)作为第一类公民，但它还包含端口和适配器接口(端口)，其中包括ORM接口、搜索引擎接口、消息传递接口等等。在我们使用命令总线和/或查询总线的情况下，这一层是命令和查询的相应处理程序所在的层。

应用程序服务和/或命令处理程序包含展开用例(业务流程)的逻辑。通常，他们的作用是:

使用存储库查找一个或多个实体;告诉这些实体做一些域逻辑;并再次使用存储库持久化实体，有效地保存数据更改。

命令处理程序可以以两种不同的方式使用:

1 它们可以包含执行用例的实际逻辑;

2 它们可以在我们的体系结构中仅仅用作连接件，接收命令并简单地触发应用程序服务中存在的逻辑。

使用哪种方法取决于上下文，例如:

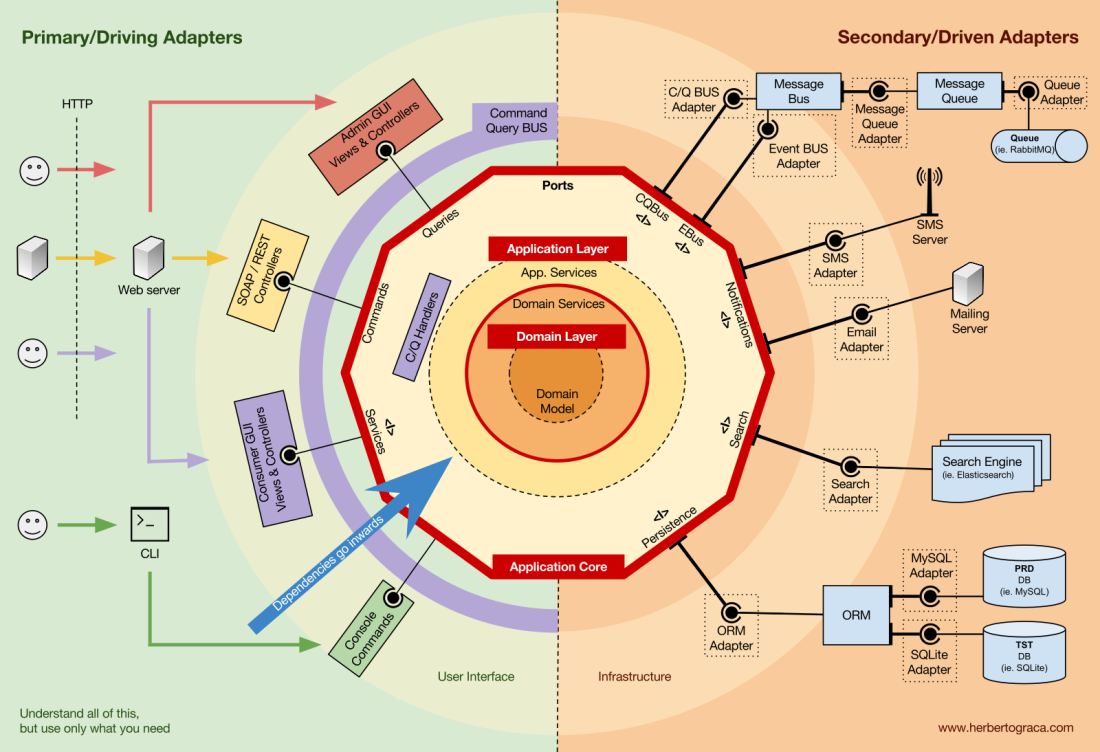
1 我们是否已经准备好了应用程序服务，现在正在添加命令总线?

2 命令总线是否允许指定任何类/方法作为处理程序，或者它们需要扩展或实现现有的类或接口?

这一层还包含应用程序事件的触发，这些事件表示用例的某些结果。这些事件触发逻辑是用例的副作用，比如发送电子邮件、通知第三方API、发送推送通知，甚至启动属于应用程序不同组件的另一个用例。

**领域层**

再往里，我们有域层。该层中的对象包含操作该数据的数据和逻辑，这些数据和逻辑是特定于域本身的，并且独立于触发该逻辑的业务流程，它们是独立的，并且完全不知道应用程序层。



**领域服务**

如上所述，应用程序服务的角色是:

使用存储库查找一个或多个实体;告诉这些实体做一些域逻辑;并再次使用存储库持久化实体，有效地保存数据更改。

然而，有时我们会遇到一些涉及不同实体的领域逻辑，相同类型的或不同类型的，我们会觉得这个领域逻辑不属于实体本身，我们会觉得这个逻辑不是它们的直接责任。

因此，我们的第一反应可能是将逻辑放在实体之外的应用程序服务中。然而，这意味着域逻辑将不能在其他用例中重用:域逻辑应该远离应用程序层!

解决方案是创建域服务，该服务的角色是接收一组实体并在其上执行一些业务逻辑。域服务属于域层，因此它对应用程序层中的类(如应用程序服务或存储库)一无所知。另一方面，它也可以使用其他域的Servi

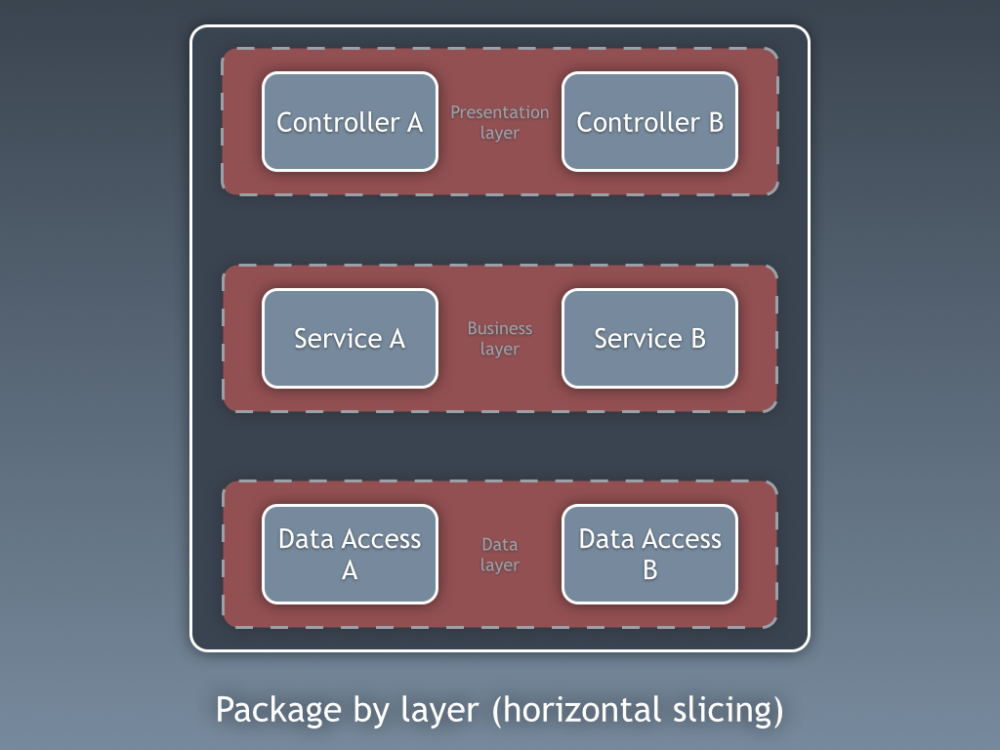
**域模型**

在最中心的是域模型，它不依赖于外部的任何东西，它包含表示域中的某些东西的业务对象。这些对象的例子首先是实体，还有值对象、枚举和域模型中使用的任何对象。

域模型也是域事件“活动”的地方。当一组特定的数据发生更改时，这些事件就会被触发，这些更改也会随之发生。换句话说，当实体发生更改时，将触发域事件，该事件携带更改的属性和新值。例如，这些事件非常适合用于事件来源。

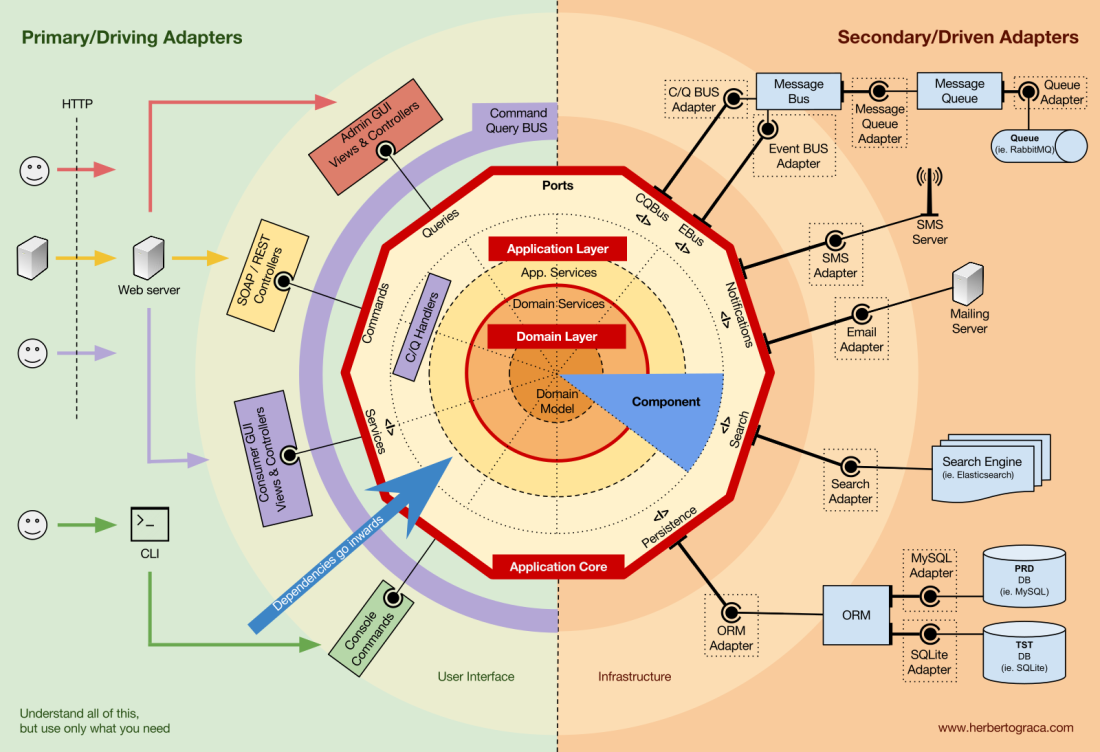
**组件**

到目前为止，我们一直基于层来隔离代码，但这是细粒度的代码隔离。代码的粗粒度分离至少同样重要，它是关于根据子域和有界上下文来隔离代码的，遵循Robert C. Martin在尖叫架构中表达的思想。这通常被称为“按特性打包”或“按组件打包”，而不是“按层打包”，Simon Brown在他的博客文章“按组件打包和与架构一致的测试”中对此有很好的解释:



我是“按组件打包”方法的倡导者，并且，从西蒙·布朗图中看到关于按组件打包的内容，我将无耻地将其更改为以下内容:

这些代码部分横切到前面描述的层，它们是我们应用程序的组件。组件的例子可以是身份验证、授权、计费、用户、审查或帐户，但它们总是与域相关的。授权和/或身份验证之类的边界上下文应该被视为外部工具，我们为其创建一个适配器并隐藏在某种端口后面。



解耦的组件

就像细粒度的代码单元(类、接口、特征、混合等)一样，粗粒度的代码单元(组件)也受益于低耦合和高内聚。

为了解耦类，我们使用依赖注入(通过将依赖注入到类中而不是在类中实例化它们)和依赖倒置(通过使类依赖于抽象(接口和/或抽象类)而不是具体类)。这意味着依赖类不知道它将要使用的具体类，它没有引用它所依赖的类的完全限定类名。

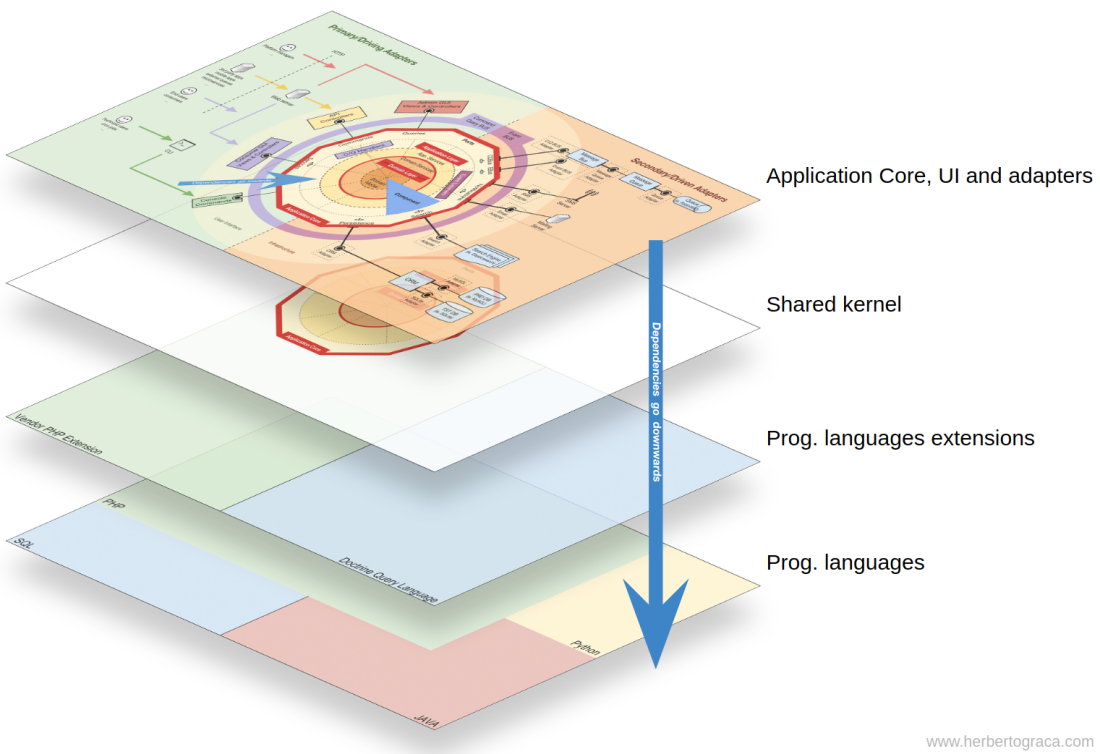
同样，完全解耦组件意味着一个组件不直接了解任何另一个组件。换句话说，它没有引用任何来自其他组件的细粒度代码单元，甚至没有接口!这意味着依赖注入和依赖倒置不足以解耦组件，我们将需要某种架构构造。我们可能需要事件、共享内核、最终一致性，甚至是发现服务!

触发其他组件中的逻辑

当我们的一个组件(组件B)需要在另一个组件(组件A)发生其他事情时做一些事情时，我们不能简单地从组件A直接调用组件B中的类/方法，因为这样A就会耦合到B。

然而，我们可以让A使用一个事件分派器来分派一个应用程序事件，该事件将被交付给任何监听它的组件，包括B, B中的事件监听器将触发所需的操作。这意味着组件A将依赖于事件分派器，但它将与B解耦。

然而，如果事件本身“存在”在A中，这意味着B知道A的存在，它与A耦合在一起。为了消除这种依赖，我们可以创建一个带有一组应用程序核心功能的库，这些功能将在所有组件之间共享，即共享内核。这意味着这些组件都将依赖于共享内核，但它们将彼此解耦。共享内核将包含应用程序和域事件等功能，但它也可以包含规范对象，以及任何有意义的共享内容，请记住，它应该像po一样最小例如，共享内核不包含事件类，而是包含事件描述(比如。名称，属性，甚至方法(尽管这些在规范对象中会更有用)在JSON这样的未知语言中，这样所有组件/微服务都可以解释它，甚至自动生成它们自己的具体实现。请阅读我的后续文章:more than concentric layers。



这种方法既适用于单片应用程序，也适用于分布式应用程序(如微服务生态系统)。然而，当事件只能异步交付时，对于需要立即触发其他组件中的逻辑的上下文，这种方法是不够的!组件将需要一个直接的HTTP调用组件b。在这种情况下,解耦的组件,我们需要发现服务,将要求它应该发送请求来启动所需的行动,或者使请求发现服务谁可以代理它t

**从其他组件获取数据**

在我看来，不允许组件更改它不“拥有”的数据，但它可以查询和使用任何数据。

**组件之间共享数据存储**

当一个组件需要使用属于另一个组件的数据时，假设一个计费组件需要使用属于accounts组件的客户端名称，计费组件将包含一个查询对象，该对象将查询该数据的数据存储。这仅仅意味着计费组件可以知道任何数据集，但是它必须通过查询的方式以只读的方式使用它不“拥有”的数据。

**每个组件分隔的数据存储**

在本例中，应用了相同的模式，但我们在数据存储级别上有了更多的复杂性。组件有自己的数据存储意味着每个数据存储包含:

1 它拥有的一组数据，并且是唯一允许改变的数据，使其成为唯一的真相来源;

2 一组数据，它是其他组件数据的副本，它不能自己更改这些数据，但组件功能需要这些数据，并且当它在所有者组件中发生更改时，需要对其进行更新。

每个组件将从其他组件创建所需数据的本地副本，以便在需要时使用。当数据在拥有它的组件中发生更改时，该所有者组件将触发一个携带数据更改的域事件。持有该数据副本的组件将侦听该域事件，并相应地更新其本地副本。

**控制流**

正如我上面所说的，控制流当然是从用户到应用程序核心，再到基础设施工具，再到应用程序核心，最后再到用户。但是这些类到底是如何组合在一起的呢?哪一个取决于哪一个?我们如何组合它们?

在Bob叔叔关于干净架构的文章之后，我将尝试用UMLish图来解释控制流程……

**没有命令/查询总线**

在不使用命令总线的情况下，控制器将依赖于应用程序服务或查询对象。

[EDIT - 2017-11-18]我完全漏掉了用于从查询返回数据的DTO，所以我现在添加了它。是谁给我指出来的吗啡镇痛药。

在上面的图中我们使用应用程序的接口服务,尽管我们可能认为这不是真正需要从应用程序服务是我们应用程序代码的一部分,我们不会想交换另一个实现,尽管我们可能完全重构它

。

查询对象将包含一个优化的查询，该查询将简单地返回一些要显示给用户的原始数据。该数据将在DTO中返回，DTO将被注入到ViewModel中。这个viewmodel可能包含一些视图逻辑，它将被用来填充一个视图。

另一方面，应用程序服务将包含用例逻辑，当我们想要在系统中做一些事情时将触发的逻辑，而不是简单地查看一些数据。应用程序服务依赖于存储库，存储库将返回包含需要触发的逻辑的实体。它还可能依赖于域服务来协调多个实体中的域流程，但这种情况很少发生。

在展开用例之后，应用程序服务可能希望通知整个系统该用例已经发生，在这种情况下，它还将依赖于一个事件调度程序来触发事件。

有趣的是，我们在持久性引擎和存储库上都放置了接口。虽然看起来有些多余，但它们的作用是不同的:

1 持久化接口是ORM上的一个抽象层，因此我们可以在不改变应用程序核心的情况下交换使用的ORM。

2 repository接口是对持久化引擎本身的抽象。假设我们想要从MySQL切换到MongoDB。持久性接口可以是相同的，而且，如果我们想继续使用相同的ORM，即使持久性适配器也会保持不变。然而，查询语言是完全不同的，所以我们可以创建新的存储库，使用相同的持久化机制，实现相同的存储库接口，但使用MongoDB查询语言而不是SQL来构建查询。

**使用命令/查询总线**

在我们的应用程序使用命令/查询总线的情况下，关系图基本上保持相同，除了控制器现在依赖于总线和命令或查询。它将实例化命令或查询，并将其传递给总线，后者将找到适当的处理程序来接收和处理命令。

在下面的图表中，命令处理程序然后使用应用程序服务。然而，这并不总是需要的，事实上，在大多数情况下，处理程序将包含用例的所有逻辑。如果我们需要在另一个处理程序中重用相同的逻辑，我们只需要从处理程序中提取逻辑到一个分离的应用程序服务中。

您可能已经注意到总线和命令、查询或处理程序之间没有依赖关系。这是因为，为了提供良好的解耦，它们实际上应该彼此不了解。总线知道什么处理程序应该处理什么命令或查询的方式应该通过配置来设置。

如您所见，在这两种情况下，所有跨越应用程序核心边界的箭头和依赖项都指向内部。如前所述，这是端口和适配器架构、洋葱架构和干净架构的基本规则。

**结论**

一如既往，我们的目标是拥有一个松耦合和高内聚的代码库，这样更改就容易、快速和安全。计划是没用的，但计划就是一切。艾森豪威尔

这意味着这些只是指导方针!应用程序是我们需要应用我们的知识的领域、现实和具体用例，它将定义实际的体系结构将是什么样子!

我们需要理解所有这些模式，但我们也总是需要思考和理解我们的应用程序需要什么，为了解耦和内聚我们应该走多远。这个决定取决于很多因素，首先是项目功能需求，但也可能包括构建应用程序的时间框架、寿命等因素

就是这样，这就是我理解这一切的方式。这是我在脑海中对它的解释。

然而，我们如何使所有这些在代码库中显式?这是我下一篇文章的主题:如何在代码中反映体系结构和领域。

最后但并非最不重要的，感谢我的同事Francesco Mastrogiacomo，帮助我使我的信息图表看起来很好。

我在一篇后续文章中进一步扩展了这些想法:more than concentric layers。