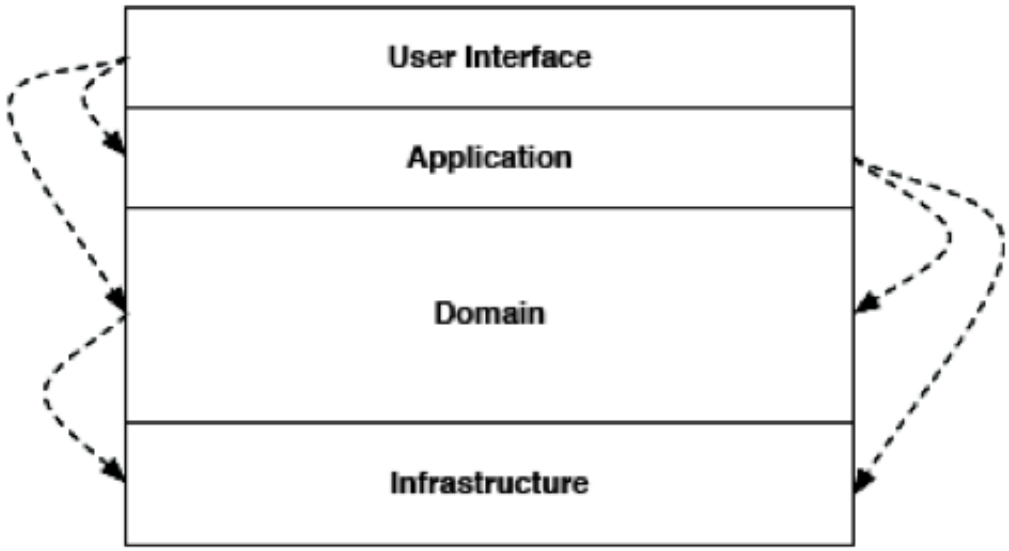
3. 架构模式与应用实践

架构模式与应用

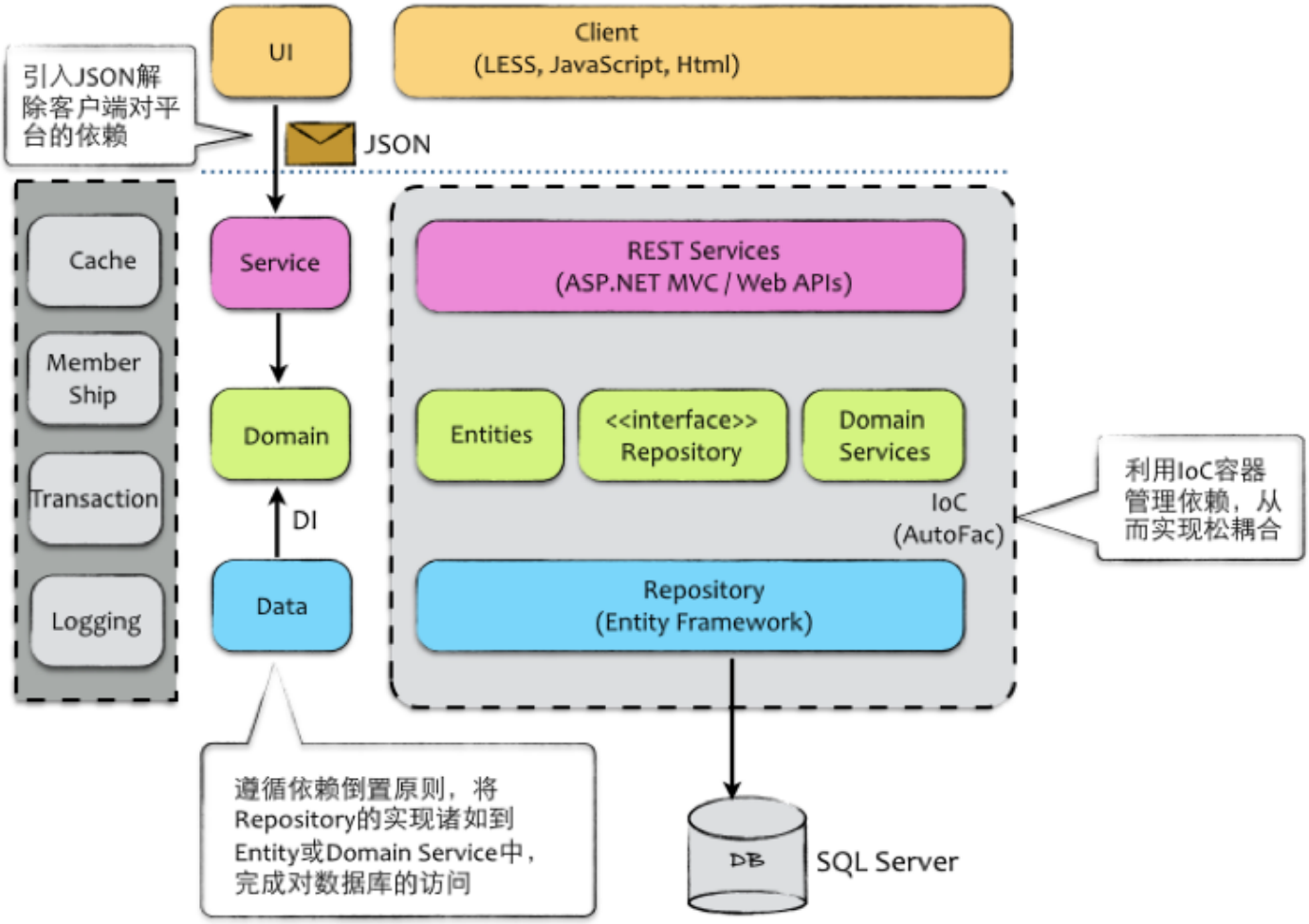
分层架构模式

在DDD（领域驱动设计）中，对传统的三层架构模式改进为四层，引入独有的Application Layer，并将数据访问层更名为Infrastructure Layer：



△ DDD分层架构

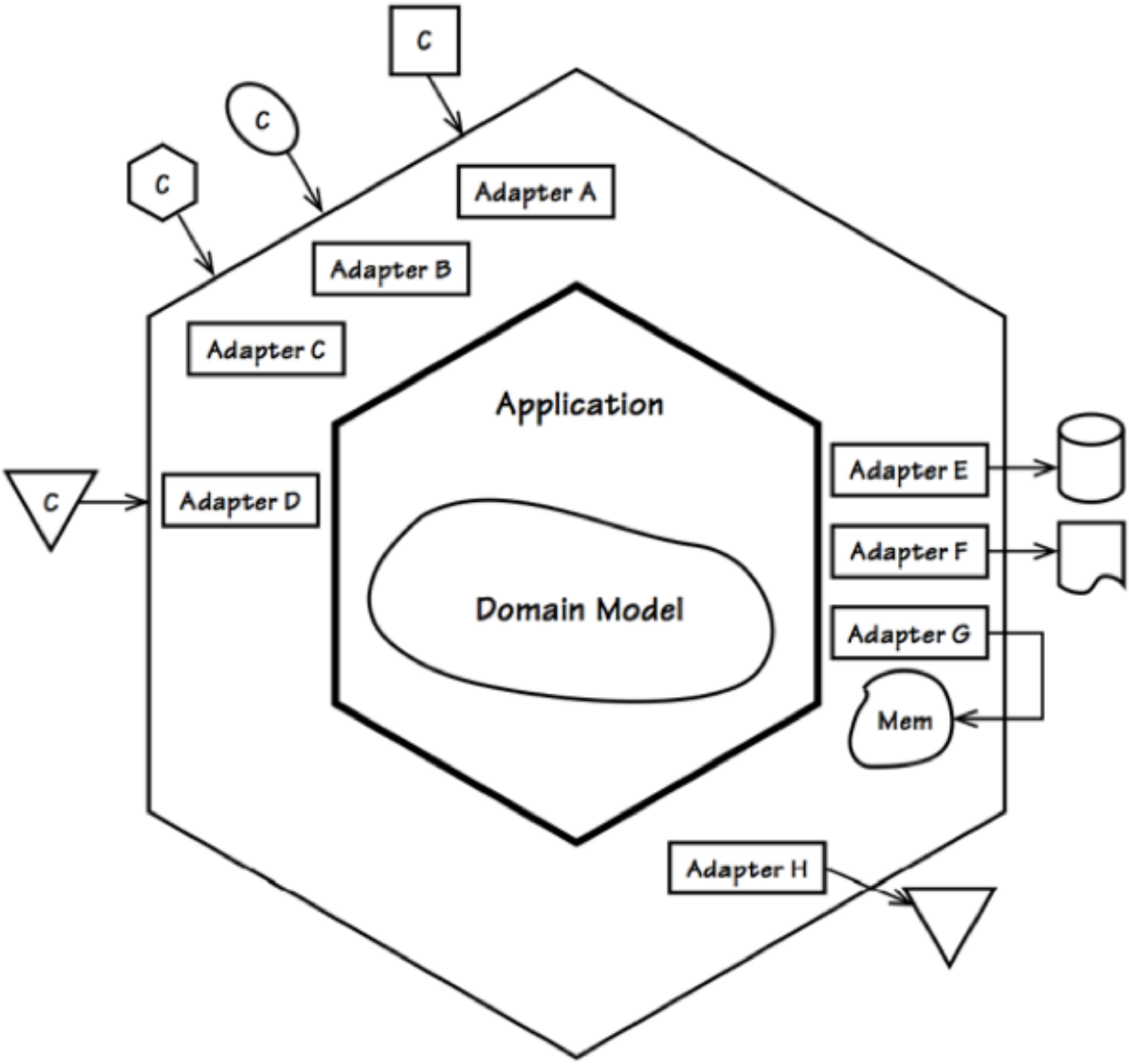
很多企业系统在设计时都是从数据驱动，从而导致开发出Martin Fowler所诟病的“贫血模型”。不过如果为领域对象引入合理的行为，则可能存在一些设计上的改进。如下是引入领域模型后在.NET平台下的一个经典分层模型：



△ 典型的.NET分层架构

六边形架构（Hexagonal Architecture）

六边形架构是Alistair Cockburn提出的一种具有对称性特征的架构风格。在这种架构中，不同的客户通过“平等”的方式与系统交互。该架构中存在两个区域，分别是“外部区域”和“内部区域”。在外部区域中，不同的客户均可以提交输入；而内部的系统则用于获取持久化数据，并对程序输出进行存储（比如数据库），或者在中途将输出转发到另外的地方（比如消息）。



△ 六边形架构（Port-Adapter）

运用六边形架构需要识别系统关注点，从架构层面（全局视角）设计，暂时可以不考虑实现细节。六边形架构这种内外分离的方式，可以有效地把系统的核心领域与边界外的基础设施隔离开，从而形成一种独立于框架，易于测试，与外部代理、UI以及数据库无关的应用架构。

CQRS架构风格

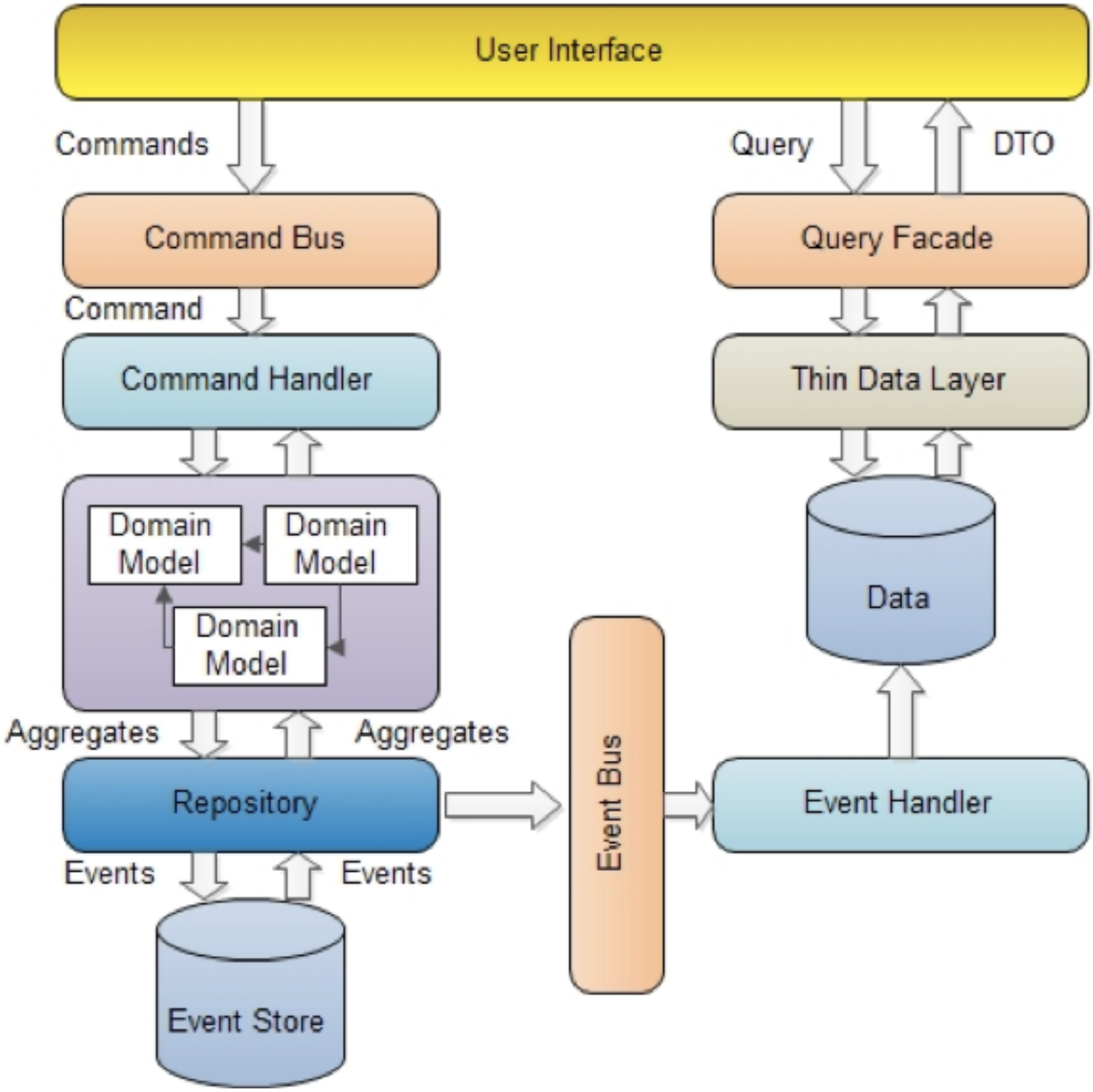
CQRS由Greg Young提出，目前在DDD领域中被广泛使用。在我看来，它甚至可以被称为是一种架构风格，可以取得与MapReduce，REST同等的地位，对软件系统的整体架构产生重要影响。

CQRS即Command Query Responsibility Seperation（命令查询职责分离），其设计思想来源于Mayer提出的CQS（Command Query Seperation）。这种命令与查询的分离方式，可以更好地控制请求者的操作。查询操作不会造成数据的修改，因而它属于一种幂等操作，可以反复地发起，而不用担心会对系统造成影响。基于这种特性，我们还可以为其提供缓存，从而改进查询的性能。命令操作则与之相反，它会直接影响系统信息的改变。查询操作与命令操作对事务的要求也不一样。由于查询操作不会改变系统状态，因而，不会产生最终的数据不一致。从请求响应的角度来看，查询操作常常需要同步请求，实时返回结果；命令操作则不

然，因为我们并不期待命令操作必须返回结果，这就可以采用fire-and-forget方式，而这种方式正是运用异步操作的前提。此外，对于大多数软件系统而言，查询操作发起的频率通常要远远高于命令操作。如上种种，都是将命令与查询进行分离的根本原因。

这就很好地阐释了我们为何需要运用CQRS模式，同时也说明了CQRS的适用场景。

只要充分理解了运用CQRS模式的意图，理解CQRS模式就变得容易了许多。下图是CQRS框架AxonFramework官方文档给出的CQRS架构图。



△

CQRS的典型架构

在这个架构图中，最核心的概念是Command、Event。以我的理解，CQRS模式的风格源头就是基于事件的异步状态机模型。抛开命令查询分离这一核心原则，这才是CQRS的基础内容。CQRS对设计者的影响，是将领域逻辑，尤其是业务流程，皆看做是一种领域对象状态迁移的过程。这一点与REST将HTTP应用协议看做是应用状态迁移的引擎，有着异曲同工之妙。这种观点（或设计视图）引出了Command与Event的概念。Command是系统中会引起状态变化的活动，通常是一种命令语气，例如注册会议RegisterToConference。至于Event，则描述了某种事件的发生，通常是命令的结果（但并不一定是直接结果，但源头一定是因为发送了

命令），例如OrderConfirmed。我发现，这种事件更接近于一种事实，即某次数据改变的结果，是一种确定无疑已经发生的事实。这一思想直接引入了Event Source，并带来Audit（审计）的好处。Event Source可以将这些事件的发生过程记录下来，使得我们可以追溯业务流程。

Command和Event都有对应的Handler来处理。它们具有一个共同的特征，即支持异步处理方式。这也是为何在架构中需要引入Command Bus和Event Bus的原因。在UI端执行命令请求，事实上就是将命令（注意，这是一个命令对象，你完全可以将其理解为Command模式的运用。注意，命令的命名一定要恰如其分地体现业务的意图）发送到Command Bus中。Command Bus更像是一个调停者（Mediator），在接收到Command时，会将其路由到准确的CommandHandler，由CommandHandler来处理该命令。在Axon Framework中，Command Bus提供了dispatch()方法对命令进行分发。也就是说，在它的实现中，并没有对Command提供异步处理，而仅仅是完成路由的功能。

Event的处理与之相似。Axon Framework同时支持同步和异步方式。从框架角度讲，提供更多的选择是一件好事。但基于CQRS模式的核心思想来看，如果对Command(包括Event)的处理未采用异步模型，它就没有发挥出足够的优势，此时采用CQRS，反而会增加设计难度，有些得不偿失。

在Command端，基本的处理流程是由UI发起命令请求，发送到CommandBus，并由它分发给对应的Command Handler来处理命令。Command Handler会与领域对象，特别是与Aggregation Root对象通信。在处理了相关的业务逻辑后，会触发Event。一方面，它会将Event放到Event Store中；另一方面，同时会将Event发送到Event Bus，再由Event Handler处理事件。Event Handler会负责更新数据源，从而保证查询端能够得到最新的数据。

然而，这一过程并不是这样简单。因为整个过程可能体现的是一个状态机。Command会导致状态的迁移，并在执行Aggregate的逻辑时，触发对应的Event。Event Handler在处理事件时，并不一定是这个业务过程的终点，它可能会发送引起下一个状态迁移的命令，从而形成一个不断迁移的过程，直至业务完全结束。这就需要我们在引入CQRS时，需要改变之前的设计思路，尽量从状态迁移的角度去理解业务逻辑。UML中的状态图是一个很好的分析工具。另外，它也带来一个挑战，就是事务。因为整个过程都涉及到数据状态的变化，当某个状态迁移出现问题时，要保证数据的最终结果是一致的。