The Clean Architecture

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

https://zhuanlan.zhihu.com/p/64343082

在过去几年我们看到关于系统架构的很多想法。这些包括：

Alistair Cockburn的六边形架构(也叫做端口与适配器)，Steve Freeman, 和 Nat Pryce在他们精彩的著作Growing Object Oriented Software(http://www.amazon.com/Growing-Object-Oriented-Software-Guided-Tests/dp/0321503627)采用。

Jeffrey Palermo的Onion Architecture

(http://jeffreypalermo.com/blog/the-onion-architecture-part-1/)

去年一片博客里的 Screaming Architecture

(http://blog.cleancoders.com/2011-09-30-Screaming-Architecture)

James Coplien 与Trygve Reenskaug的

DCI(http://www.amazon.com/Lean-Architecture-Agile-Software-Development/dp/0470684208/)。

Ivar Jacobson的书： Object Oriented Software Engineering: A Use-Case Driven Approach 的BCE（http://www.amazon.com/Object-Oriented-Software-Engineering-Approach/dp/0201544350）

尽管这些架构在一些细节上都有不同，它们仍是相似的。他们都有同样的目标，隔离关注点。他们都通过将软件分层来达到隔离。每个都至少有一层业务规则，另一层作为接口。

每个这些架构产出的系统都是：

独立的框架。架构不依赖一些存在类库的特性。这样你可以像工具一样使用这种框架，而不需要让你的系统受到它的约束条件。

可测试。业务规则可以脱离UI，数据库，web服务器或其他外部元素进行测试。

独立的UI。UI可以很容易的更换，系统的其他部分不需要变更。例如，Web UI可以被换成控制台UI，不需要变更业务规则。

独立的数据库。你可以交换Oracle或SQL Server，用于Mongo，BigTable，CouchDB或其他的东西。你的业务规则不与数据库绑定。

独立的外部代理。实际你的业务规则并不知道关于外部世界的任何事情。

这篇文章上面的图试着将以上所有架构整合成一个可执行的想法。

依赖规则

同心圆表示软件的不同部分。大体上，你走的越远，软件的级别更高。外部的圆是机制，内部的圆是策略。

让这个架构工作的覆盖规则是依赖规则。这个规则说明了源代码依赖只能向内。内部圆不能知道任何外部圆的事。实践中，外部圆里一些声明的名字不能被内部圆里的代码提到。这包括，函数，类，变量或其他任何软件实体。

同样的，外部圆使用的数据格式不应该被内部圆使用，尤其是当这些格式是被外部圆使用的框架生成的时候。我们不想让外部圆的东西影响到内部圆。

实体

实体封装企业域范围的业务规则。实体可以是一个有方法的对象，也可以是一组数据结构和函数。只要企业里不同的应用可以使用这些实体就可以。

如果你不是企业级，而只是写一个单体应用，那么这些实体就是应用的业务对象。它们封装了最通用和高层的规则。当外部变化时它们基本不太会变化。例如，你不会认为这些对象会因为页面导航或安全方面的变化而改变。任何特定应用的操作都不应该影响实体层。

用例

这层的软件包含特定应用的业务规则。它封装并实现了系统的所有用例。这些用例组织了实体中的数据流向，并指挥这些实体使用他们的企业域业务规则来完成用例的目标。

我们不期望这层影响实体。我们也不希望这层会在如数据库，UI，或其他常用框架这样的外部变化时被影响。这层隔离了以上关注点。

当然我们期望对于应用操作的变化会影响用例而进一步影响到这层的软件。 如果一个用例的细节变化了，那么这层的代码肯定也会被影响。

接口适配器

这层的软件是一组适配器，其将数据转换成从用例和实体最合适的格式，到对于一些类似数据库或网站这种外部设施最合适的格式。在这一层，举个例子，会包含GUI的MVC架构。Presenters, Views,与Controllers都属于这里。模型基本就是从controllers传递到用例的数据结构，并从用例返回到presenters和views。

类似的，数据被转换了，在这层，从对于实体和用例合适的结构，变成对于持久层框架使用的结构。这圈内的代码不应该知道数据库。如果数据库是一个SQL数据库，那么所有SQL都应该在这层内，特别是此层与数据库有关的部分。

这层其他适配器也需要将数据从类似外部服务的外部的结构，转换成用例和实体使用的内部结构。

框架与驱动

最外层主要组合了数据库，网络框架这样的框架和工具。在这层你除了写一些与内层环通信的胶水代码，基本不会有其他代码。

这层是所有细节存在的地方。网络是细节。数据库是细节。 我们将这些东西放在外部保证它们不会影响其他部分。

只有四个圈？

不是的，圆圈是个示意。你可能发现你需要不止4个。没有规则说你一定要有四个。 实际上，依赖规则一直存在。源代码依赖一直指向内部。当你向内部移动时抽象的层次在增加。最外部的圆是很低层的具体细节。当你内移时软件变得更抽象，并封装了高一级的策略规则。最内部的圆是最普遍的抽象层级。

跨越边界

在图的右下方是我们穿越圆圈边界的示例。它展示了Controller和Presenter与下一层的用例进行通信。注意控制流。它从controller出发，穿过用例，然后在presenter里执行。也注意下源码依赖。它们每个都指向内部的用例。

我们通常使用依赖反转原则解决这个明显的问题。在java这样的语言中，我们会整理源码依赖与控制流相反的接口和继承关系，让它们从边界正确的穿过。

例如，用例需要调用presenter。但是，这个调用不能直接进行因为会违反依赖规则。外圈的名字不能被内圈提到。所以我们的用例调用内圈的一个接口（在这个例子里是Use Case Output Port），并让外圈的presenter实现它。

架构里所有的边界穿越都用这个技巧。我们使用动态多态来创建与控制流相反的源码依赖，以便于无论在控制流的任何方向都不会违反依赖规则。

什么样的数据会穿越边界

正常来说穿过边界的数据是简单数据结构。你可以使用基本结构或简单的Data Transfer 对象。或者可以方便的进行函数赋值的数据。或者你可以打包进一个hashmap，或者将它组装成一个对象。重要的是穿过边界的是隔离，简单的数据结构。我们不想搞变通传递实体或数据库行数据。我们不想数据结构有任何违反依赖规则的依赖。

例如，很多数据库框架在查询后返回一个方便的数据格式。我们可以叫它RowStructure(行结构)。我们不想将这个行机构通过边界传递给内部的圈。这会导致内部圈需要知道外部圈的内容进而违反依赖规则。

所以当我们在边界传递数据是，要注意其应该是内部圈的格式。

结论

遵从这些简单规则并不难，并且能帮你减少以后的问题。通过将软件隔离分层，并遵从依赖规则，你可以建立一个真正可测试的系统，包含了以上所有好处。当任何系统额外部部分过时了，比如数据库或web框架，你可以容易的替换这些过时的元素。