<https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/>

由Alistair Cockburn在2005年记录的六边形架构是一种软件架构，它有很多优点，自2015年以来又重新引起了人们的兴趣。

六角形建筑的初衷是:

允许应用程序同样由用户、程序、自动化测试或批处理脚本驱动，并在与最终运行时设备和数据库隔离的情况下进行开发和测试。

为了探索通过自动化测试或独立于数据库开发和测试应用程序的好处，我们建议您阅读我们最近发布的关于测试金字塔的系列博文:实践中的测试金字塔。

这个承诺非常有吸引力，而且它还有另一个好处:它允许隔离应用程序的核心业务，并独立于其他一切自动测试其行为。这可能是这个体系结构吸引领域驱动设计(DDD)实践者的原因。但是要注意，*DDD和六角形结构是两个截然不同的概念，它们可以相互加强，但不一定一起使用。但这是另一个话题了!*

最后，这个体系结构的设置并不复杂。它基于一些简单的规则和原则。让我们探讨一下这些原则，看看它们在实践中意味着什么。

1 六边形架构的原理

2 细节:代码内部和外部是如何组织的?

3 Detail: At the Runtime

4 Detail: 依赖倒置在右边

5 Detail: 为什么界面在左边?

6 测试六角形架构

7 为了进一步推进

8 References

一 、六边形架构的原理

六边形建筑基于三个原则和技术:

显式分离用户端、业务逻辑和服务器端

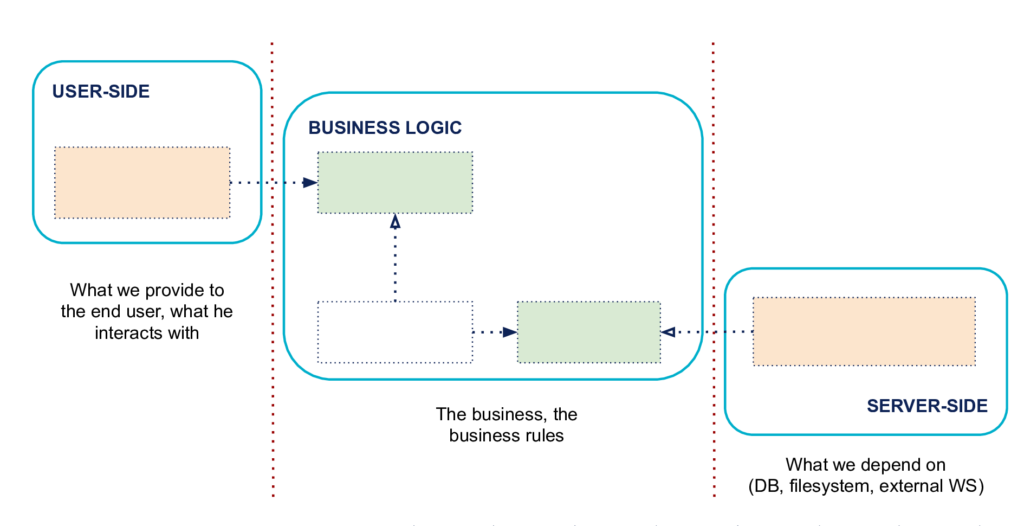
依赖关系从用户端和服务器端转移到业务逻辑

我们通过使用端口和适配器来隔离边界

词汇表注意:在本文的其余部分，将使用“用户端”、“业务逻辑”和“服务器端”这几个词。这些词来自原始文章，在下一节中定义。

**1 原则:分开用户端、业务逻辑和服务器端**

第一个原则是显式地将代码分为三个形式化的大区域。



左边是用户端

这是用户或外部程序与应用程序交互的端。它包含允许这些交互的代码。通常，您的用户界面代码、API的HTTP路由、到使用应用程序的程序的JSON序列化都在这里。

这是我们找到驱动业务逻辑的参与者的地方。

注意:Alistair Cockburn也称其为左侧。

业务逻辑在中间

他是我们想要从左右两边分离出来的部分。它包含所有涉及和实现业务逻辑的代码。业务词汇表和与解决应用程序的具体问题相关的纯业务逻辑，使其丰富和特定的一切都处于中心位置。理想情况下，不知道如何编写代码的领域专家可以阅读这一部分的代码，并指出不一致的地方(这是真的，这些事情可能会发生在你身上!)

注:Alistair Cockburn也称其为中心。

右边是服务器端

在这里，我们将发现应用程序需要什么，它推动什么工作。它包含必要的基础设施细节，比如与数据库交互的代码、调用文件系统的代码，或者处理对您所依赖的其他应用程序(例如)的HTTP调用的代码。

这是我们找到由业务逻辑管理的参与者的地方。

注:Alistair Cockburn也称其为右侧。

以下原则将允许在用户端、业务逻辑和服务器端之间实现逻辑分离。

为什么这很重要?

这种分离的第一个重要特征是它分离了问题。在任何时候，您都可以选择专注于单个逻辑，几乎独立于其他两个逻辑:用户端逻辑、业务逻辑或服务器端逻辑。它们在不混合的情况下更容易理解，而且每个逻辑的约束对其他逻辑的影响较小。

另一个特点是我们将业务逻辑放在代码的最前面。它可以被隔离在一个目录或模块中，以便对所有开发人员显示。它可以被定义、改进和测试，而无需承担其他工作的认知负荷

最后，在自动化测试方面(我们将在下面看到)，我们将通过合理的努力成功地测试:

整个业务逻辑，

独立于服务器端的用户端和业务逻辑之间的集成

在用户端独立集成业务逻辑和服务器端

**说明:应用程序的一个小示例**

为了更具体地说明这些原则，我们将使用在2017年由Thomas Pierrain (@tpierrain)和Alistair Cockburn (@TotherAlistair)自己提出的“Alistair in the Hexagon”事件中使用的小例子。注意:您将在本文的末尾找到视频和事件代码。

这个小应用程序的目的是提供一个命令行程序，将诗歌写入控制台的标准输出。

这个应用程序的预期输出示例:

$ ./printPoem

Here is some poem:

I want to sleep

Swat the files

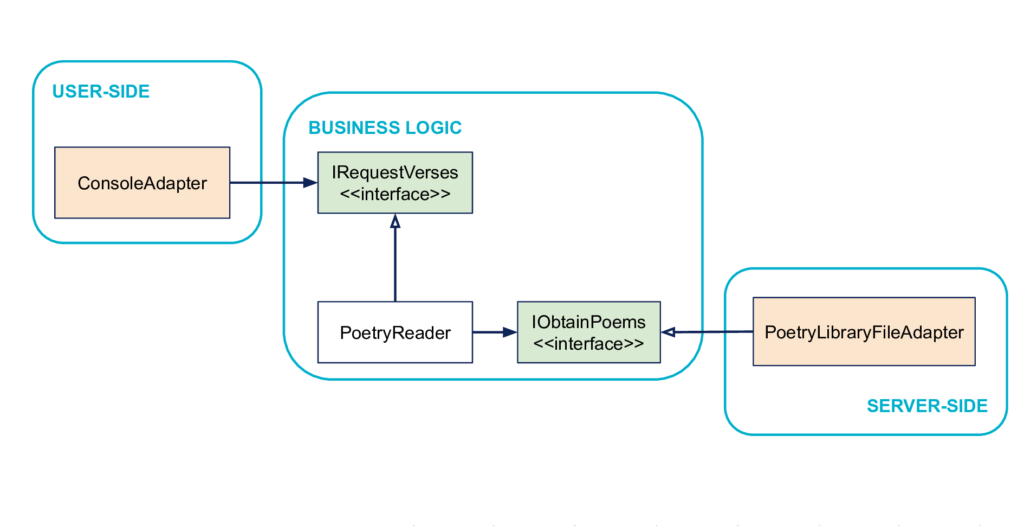
Softly, please.

-- Masaoka Shiki (1867 - 1902)

Type enter to exit...

为了正确地说明这三个区域(用户端、业务逻辑、服务器端)，该应用程序将在一个外部系统(文件)中搜索诗歌。我们也可以将这个应用程序连接到数据库，原理是相同的。进入翻译页面

在这方面，我们如何应用这一第一原则，即分成三个区域?如何在左边(什么驱动)，中间(核心业务)和右边(什么驱动)进行分配?



客户端

从用户的角度来看，这个程序是作为控制台应用程序来呈现的。所以主机的概念是在用户的左边。用户将通过控制台来驱动域。

服务器端

从技术上讲，在我们的例子中，诗歌存储在一个文件中。这个文件的概念将在服务器端右侧找到。该业务将通过导航右侧来发出它的诗歌请求，具体由一个PoetryLibraryFileAdapter实现。

在这里，正如上面所提到的，我们可以轻松地交换诗歌的来源(文件、数据库、web服务……)。源文件的实际实现因此是一个技术细节(也称为技术实现细节)。

他的业务逻辑

在这种情况下，我们的核心业务，对用户来说有价值的是读诗的概念。例如，我们可以在代码中用一个PoetryReader类具体化这个概念

用户端→业务逻辑交互

从业务角度来看，请求是否来自控制台应用程序并不重要，这是我们希望能够抽象的技术细节。这正是最初的意图之一:“由用户和测试驱动”。因此在业务逻辑中没有控制台的概念。然而，从用户的角度来看(=它提供的服务)，我们的应用程序所允许的是请求诗歌。我们将在业务逻辑(由irequestverse具体化)中找到这个概念，它将允许用户端与业务逻辑交互。

业务逻辑→服务器端交互

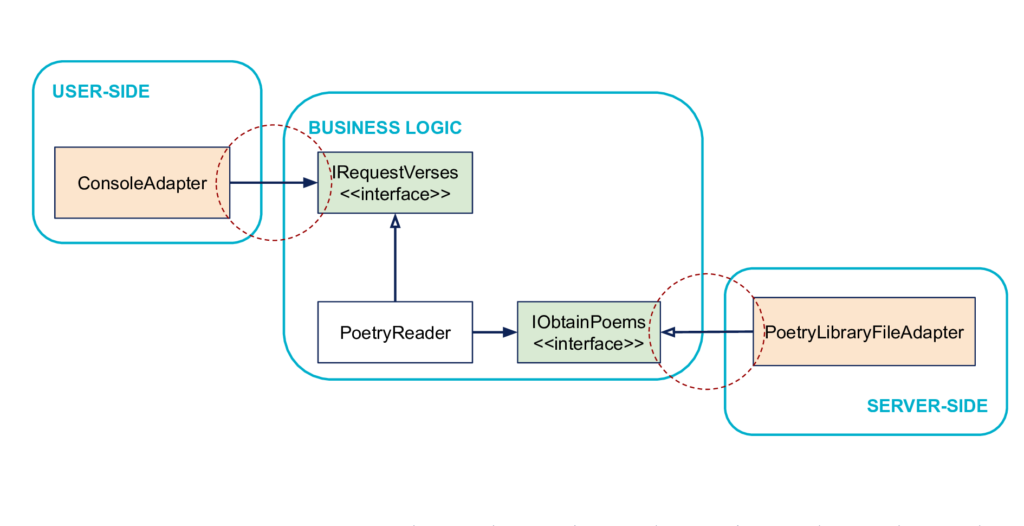
同样，从业务逻辑的角度来看，无论诗词来自文件还是数据库，我们都希望能够独立于外部系统测试我们的应用程序。业务逻辑中没有文件的概念。要操作，域名仍然需要获得诗歌。我们在业务逻辑中发现了以IObtainPoems接口的形式获取poem的概念。获得诗词的概念将允许域与服务器端交互。

注意:从这里开始，当您阅读图表时，您可以开始观察显示类之间关系的箭头。实箭头表示调用或组合交互。没有填充的箭头表示继承关系(如在UML中)。但不需要马上分析所有的东西，我们稍后将详细探讨它。

注意:名称IRequestVerses和IObtainPoems表示许多接口，我们将按照一个原则讨论它们。在这个轶事中，接口名称以“i”开头的惯例已经不再流行，但Thomas Pierrain将接口名称解读为第一人称单数的句子。IRequestVerses读到:我请求verses为例。我喜欢这个想法。

**原则:依赖关系进入内部**

这是实现这一目标的基本原则。我们已经在前面的原则中看到了这一点。



**原则:依赖关系归于业务逻辑**

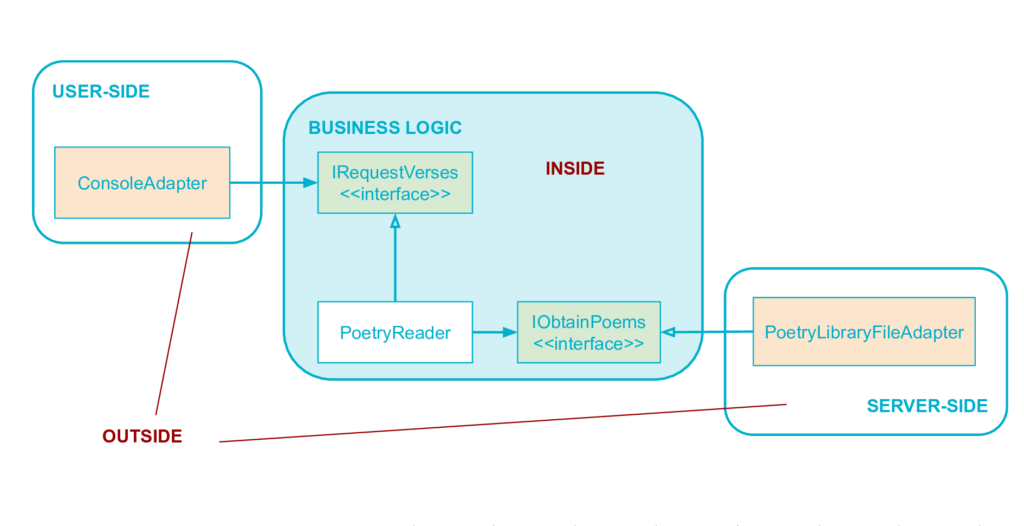
程序可以由控制台和测试控制，业务逻辑中没有控制台的概念。业务逻辑并不依赖于用户端，而是依赖于业务逻辑的用户端。用户端(ConsoleAdapter)依赖于诗歌请求的概念irequestverse(在用户方面定义了一个通用的“诗歌请求”机制)。

类似地，程序可以独立于外部系统进行测试，业务逻辑不依赖于服务器端，相反。服务器端依赖于业务逻辑，通过获取诗词的概念iobtain诗词。从技术上讲，服务器端类将继承业务逻辑中定义的接口并实现它，我们将在下面详细讨论依赖倒置。

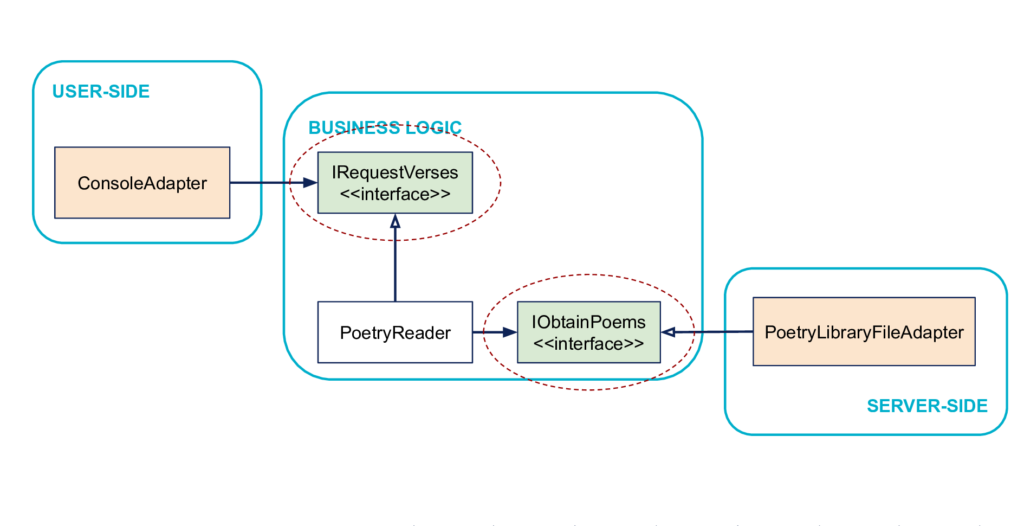
内部和外部

如果我们把依赖关系(<<依赖于…>>)看作箭头，那么这个原则将中心业务逻辑定义为内部，其他的都定义为外部(见图)。当我们讨论六边形建筑时，我们经常会发现这些内部和外部的概念。它甚至可以是需要记住和传递的基本要点:依赖关系进入内部。

换句话说，一切都取决于业务逻辑，业务逻辑不依赖任何东西。Alistair Cockburn坚持内部和外部的划分，这比用户端和服务器端之间的区别更加结构化，以解决最初的问题。



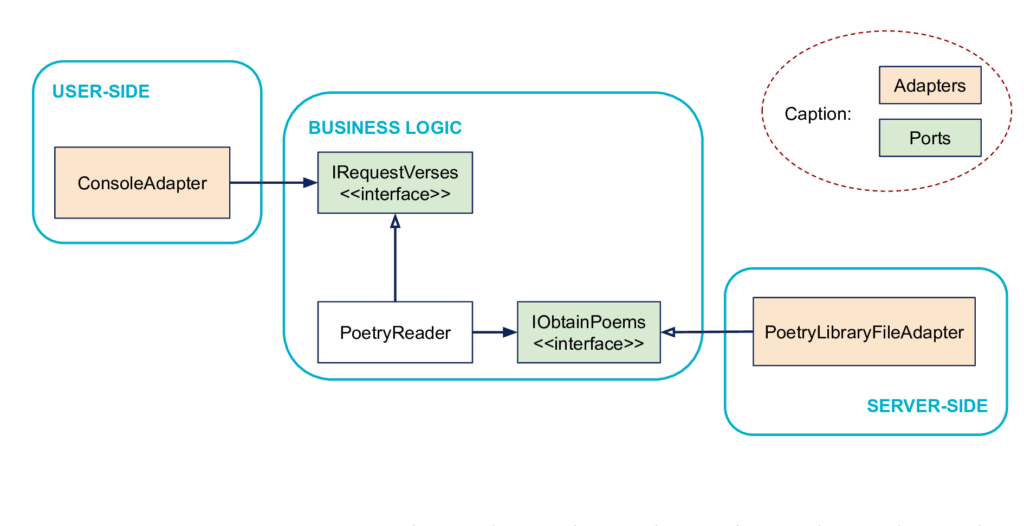
**原则:边界与接口相互隔离**



总之，用户端代码通过业务代码中定义的接口(这里是irequestverse)驱动业务代码。业务代码通过业务代码中定义的接口驱动服务器端(IObtainPoems)。这些接口作为内外之间的显式绝缘体。

一个比喻:端口和适配器

六边形架构使用端口和适配器的隐喻来表示内外之间的交互。业务逻辑定义了端口，如果各种适配器遵循端口定义的规范，它们可以在端口上互换连接。

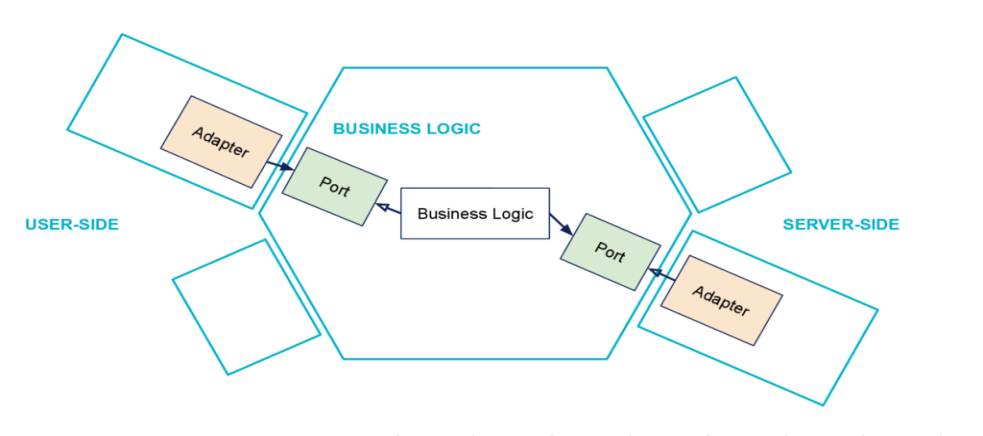


例如，我们可以想象业务逻辑的一个端口，在单元测试期间，我们将在其上连接硬编码的数据源，或者在集成测试中连接真实的数据库。只需在服务器端编写相应的实现和适配器，业务逻辑不会受到此更改的影响。

这些由业务代码定义的接口是端口和适配器隐喻的端口，它们隔离并允许与外部世界进行交互。注意:如前所述，端口是由业务定义的，因此它们在内部。

另一方面，代表外部代码的适配器在端口和其他用户端代码或服务器端代码之间形成粘合剂。在这里，适配器分别是ConsoleAdapter和PoetryLibraryFileAdapter。这些适配器在外面。

另一个比喻:六边形



这个建筑的另一个隐喻是六边形，正如我们在前面的图中看到的。为什么一个六边形?主要原因是它是一种易于绘制的形状，为表示图上的多个端口和适配器留出了空间。事实证明，即使六边形最终只是一种传闻，六边形架构的表达也比端口和适配器模式更受欢迎。可能是因为听起来更好?  
理论部分已经结束，没有其他原则:对于其他一切，我们都是完全自由的。

1. 细节:代码内部和外部是如何组织的?

除了上面看到的原则之外，我们完全可以自由地按照我们想要的方式组织每个区域内的代码。

关于业务代码内部，一个好主意是选择根据业务逻辑来组织它的模块(或目录)。

要避免的一种组织是按类型对类进行分组。例如，“ports”目录，或“repositories”目录(如果您使用此模式)，或“services”目录。在您的业务代码中考虑100%的业务，包括模块或目录的组织!理想的情况是能够打开一个目录或业务逻辑模块，立即了解您的程序解决的业务问题;而不是只看到“存储库”、“服务”或其他“管理器”目录。

另见此主题:

<https://medium.com/@msandin/strategies-for-organizing-code-2c9d690b6f33>

<https://martinfowler.com/bliki/PresentationDomainDataLayering.html>

1. Detail: At the Runtime

如何实例化所有这些来满足运行时依赖性?如果你正在使用依赖注入框架，你可能不需要问自己这个问题。但我认为，要理解六角形架构，看看应用程序启动时发生了什么是很有趣的。为了做到这一点，至少在本文中不要使用依赖注入框架。

例如，如果我们手工实例化所有东西，下面是我们如何编写应用程序的入口点:

class Program

{

static void Main(string[] args)

{

// 1. 实例化右侧适配器 ("go outside the hexagon")

IObtainPoems fileAdapter = new PoetryLibraryFileAdapter(@".\Peoms.txt");

// 2. 实例化六角

IRequestVerses poetryReader = new PoetryReader(fileAdapter);

// 3. 实例化左侧适配器 ("I want ask/to go inside")

var consoleAdapter = new ConsoleAdapter(poetryReader);

System.Console.WriteLine("Here is some...");

consoleAdapter.Ask();

System.Console.WriteLine("Type enter to exit...");

System.Console.ReadLine();

}

}

实例化顺序通常是从右到左:

1 首先，我们实例化服务器端，这里是将读取文件的fileAdapter。

2 我们实例化将由应用程序驱动的业务逻辑类，poetryReader通过注入构造函数将fileAdapter注入其中。

3 安装用户端，即将驱动poetryReader并写入控制台的consoleAdapter。这里，poetryReader通过注入构造函数注入到consoleAdapter中。

我们说过内部不应该依赖于外部。那么，为什么我们要将fileAdapter(来自服务器端的代码)注入poetryReader(来自业务逻辑的代码)呢?

之所以这样做，是因为通过查看模式和代码，fileAdapter除了是一个PoetryLibraryFileAdapter(服务器端)之外，还继承了IObtainPoems的一个实例。

在实践中，PoetryReader并不依赖于PoetryLibraryFileAdapter，而是依赖于IObtainPoems，后者在业务逻辑代码中得到了很好的定义。您可以通过查看其构造函数的签名来检查它。

public PoetryReader(IObtainPoems poetryLibrary)

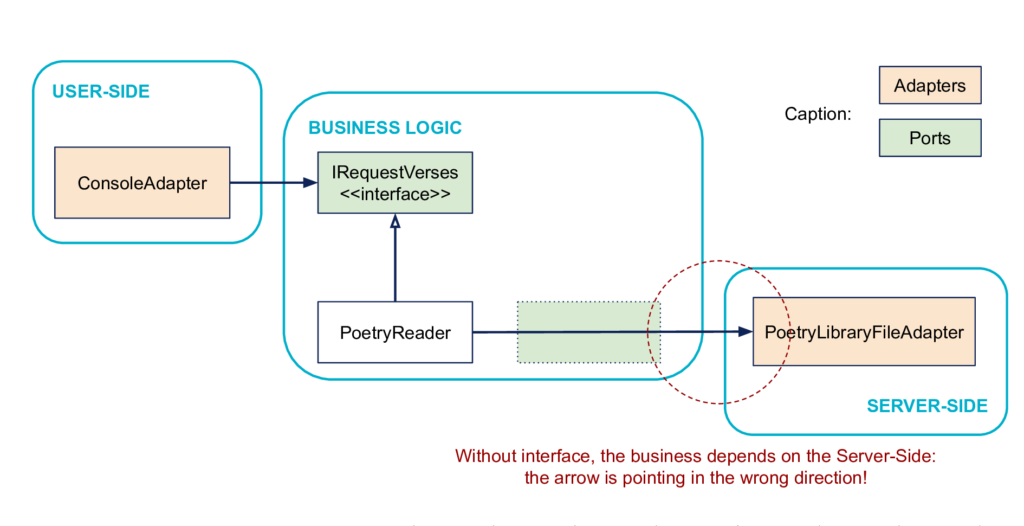
{ this.poetryLibrary = poetryLibrary;}

PoetryLibraryFileAdapter和PoetryReader是弱耦合的。

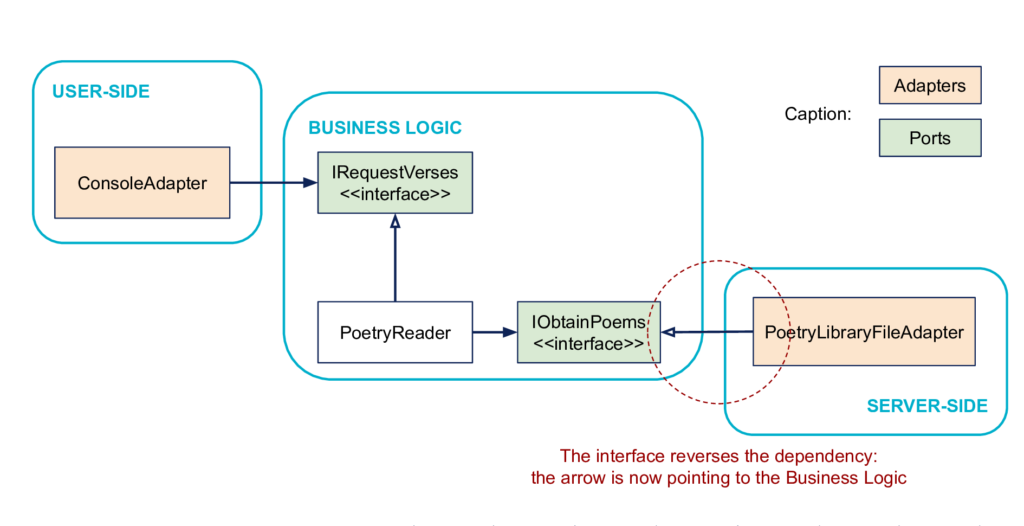
1. Detail: 依赖倒置在右边

fileAdapter依赖于业务的定义(这里的依赖是通过继承实现的)，但在运行时，poetryReader实际上可以控制fileAdapter的实例，这是依赖倒置的典型例子。

实际上，如果没有IObtainPoems接口，业务代码将依赖于服务器端代码的定义，这是我们希望避免的:



该接口允许反向此依赖的方向:



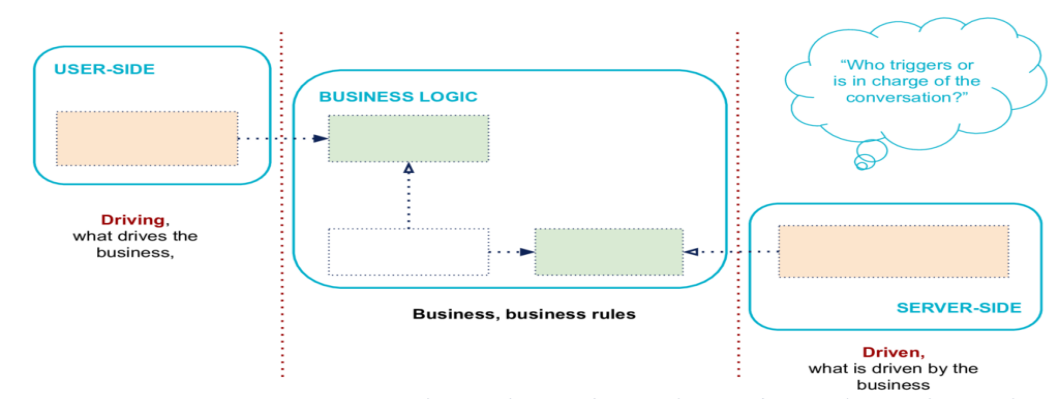
除了使业务独立于外部系统之外，右边的这个接口还允许满足著名的D SOLID，即依赖倒置原理。这一原则说:

1 高层模块不应该依赖于低层模块。两者都必须依赖于抽象。

2 抽象不应该依赖于细节。细节必须依赖于抽象。

如果我们没有接口，我们将有一个高级模块(业务逻辑)，它将依赖于低级模块(服务器端)。

注意:对于左侧代码和业务代码之间的交互，依赖关系自然是向右的。



交互实现中的这种差异与用户端/业务逻辑和业务逻辑/服务器端关系之间的差异有关。提示:用户端驱动业务逻辑，服务器端驱动业务逻辑。

1. Detail: 为什么界面在左边?

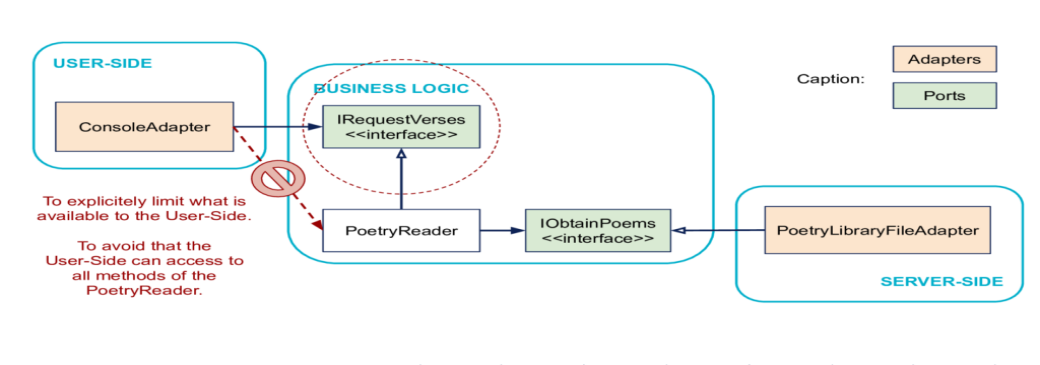
因为用户端和业务逻辑之间的依赖关系已经在正确的方向上了，所以irequestverse接口的作用不是逆转依赖关系。

然而，它仍然有一个兴趣:显式地限制用户端代码和业务逻辑代码之间的耦合面。

在实践中，PoetryReader类可以拥有irequestverse接口之外的其他方法。重要的是，ConsoleAdapter不知道这一点。

它与另一个SOLID原理相一致，即接口分离原则（Interface Segregation Principle）

不应该强迫客户端依赖于他们不使用的方法。



但是一旦你理解了意图，如果左边的端口只有一个方法，它的实现也只有一个方法，就像我们的例子一样，这个接口真的有必要吗?在一种动态语言中，最终将通过duck typing工作?

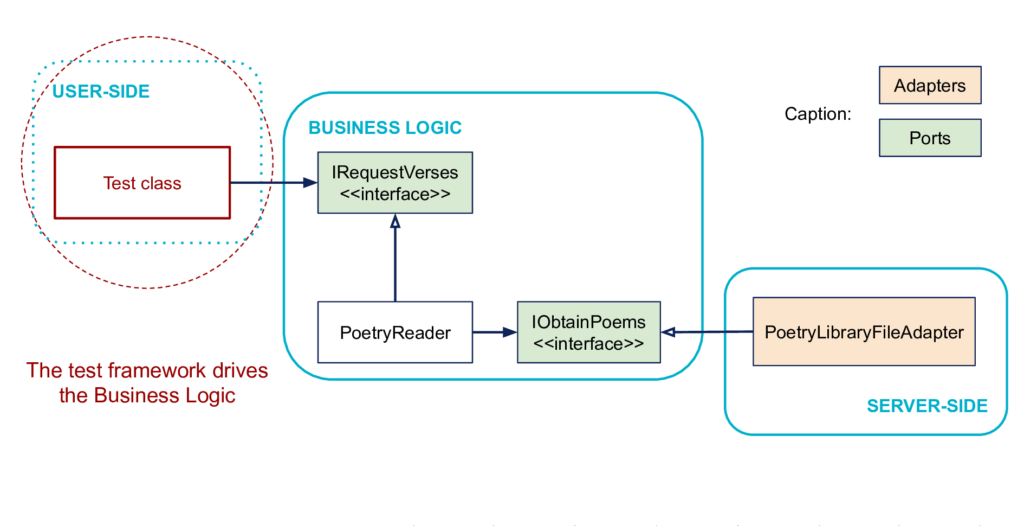
我们可以用一个问题来回答:你的团队对此怎么看?是否每个人都清楚隔离的目标，甚至不需要一个接口来触发对话?这完全由你来决

1. 测试六角形架构

这个软件架构的一个重要好处是它简化了测试自动化，这是它最初的意图的一部分。

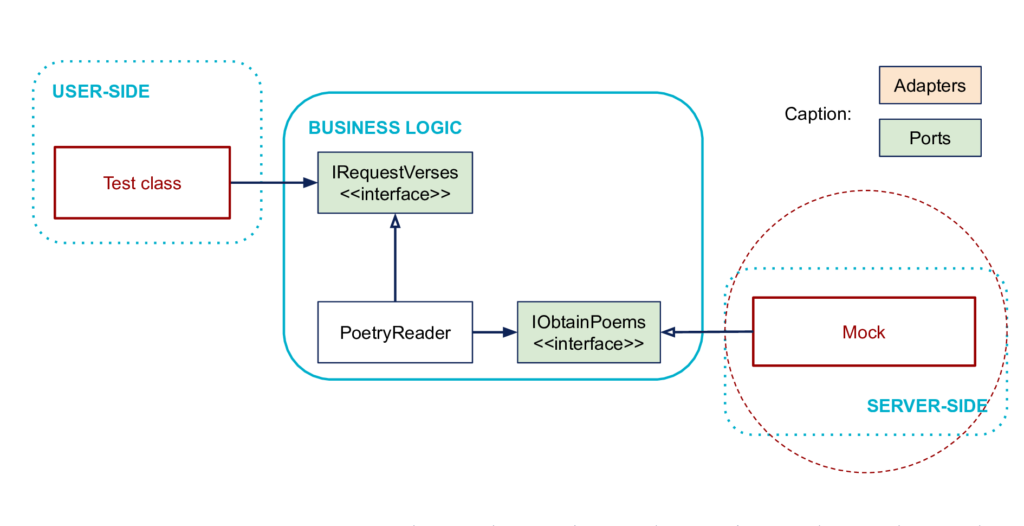
如何从用户端替换一些代码?

一般情况下，测试框架可以直接扮演左边代码的角色。实际上，测试代码可以直接驱动业务逻辑代码。



如何替换一些服务器端代码?

右边的代码必须由业务驱动。一般来说，如果你想要编写一个单元测试，你可以根据你想要测试的内容用模拟或任何其他形式的测试double来替换它。



目标达成!

允许应用程序由用户、程序、自动化测试或批处理脚本驱动，并在与可能的执行系统和数据库隔离的情况下开发和测试。

小心!这并不妨碍您测试用户端和服务器端代码，任何代码都值得测试。在这个问题上，我再次建议您参考“实践中的测试金字塔” https://blog.octo.com/en/the-test-pyramid-in-practice-1-5/系列。

事实上，通过结合我们所替换的或不替换的，我们看到，通过这个架构，我们可以测试我们想要的:

整个业务逻辑，

独立于服务器端的用户端和业务逻辑之间的集成

在用户端独立地集成业务逻辑和服务器端

1. 为了进一步推进

作为一个团队来说，谁已经知道在家里怎么做了?

继续，在实际生活中，在您的代码上进行实验。例如，一个小型的个人项目，或者你的团队的一个小型项目。什么对你来说是容易的，什么是困难的?

以下是一些在执行过程中可能会遇到的额外问题:

1 一个端口只能有一个方法，也可以将多个方法分组。对你来说，什么是合理的?

2 即使它很好地遵循了依赖原则，也不一定要将代码划分为三个显式的模块、目录、包或名称空间。在Thomas Pierrain的代码中，我多次看到“域”目录中的业务逻辑代码，以及“基础设施”目录中的用户端和服务器端代码。在他的例子中，内部代码位于六边形。域命名空间，以及六边形的外部代码。下文名称空间。

提醒一下:世上没有灵丹妙药。六边形建筑是复杂性和力量之间的一个很好的妥协，也是发现我们已经解决的主题的一个很好的方式。但这只是众多解决方案中的一个。对于简单的情况，它可能太复杂，对于复杂的情况，它可能太简单。还有其他一些软件架构值得探索。例如， Clean Architecture在正规化和绝缘方面走得更远(SOLID)。或者在一个不同但兼容的轴上，CQRS可以更好地分离读取和写入

1. References

“阿利斯泰尔在六边形”活动的视频在这里。这次活动的代码在Thomas Pierrain的github上。  
你也可以阅读这些关于这个主题的好文章:

更新注意:在本文的第一个版本中，我们使用了“应用程序”、“领域”和“基础设施”来代替“用户端”、“业务逻辑”和“服务器端”。我们回到最初的词，因为这种替换有时是模棱两可的，没有帮助。