**依赖注入技术Dependency Injection**

**定义：**

**维基百科Wikipedia**：

在软件工程中，依赖项注入是一种技术，其中一个对象接收它所依赖的其他对象，称为依赖项。通常，接收对象被称为客户端，而传入的（“对象”）对象则被称为服务。将服务传递给客户端的代码称为注入器。注入器不指定客户端将使用哪个服务，而是告诉客户端使用什么服务。“注入”是指将依赖项（服务）传递给使用它的客户端。

**Mark Seeman**：

依赖注入是一组软件设计原则和模式，使我们能够开发松散耦合的代码。

**松散耦合代码的好处**：

**可扩展性**

可扩展性是一种能够轻松地向代码中添加新功能的能力。“容易”的部分意味着我们可以在完全隔离的区域进行更新，而不是需要在整个代码库中更新不同的部分。

**后期绑定**

如前所述，后期绑定是选择在运行时使用的组件而不是编译时的能力。我们只有在我们的代码是松散耦合的情况下才能这样做——我们的代码只关心抽象，而不是一个特定的具体类型。这允许我们交换组件，而不需要修改我们的代码。

**并行开发**

如果我们的代码是松散耦合的，那么多个开发团队就可以更容易地处理同一个项目。我们可以让一个团队在业务层工作，还有另一个团队在服务层工作。因为这些层是独立的，所以团队将在不同的源文件中，它们不会直接影响彼此。

**可维护性**

当我们的组件是独立的时，功能就是孤立的。这意味着，如果我们需要查找或调整bug的功能，我们就知道去哪里寻找。

**可测试性**

单元测试是一个非常重要的课题。单元测试的主要目标是孤立地测试小段代码。当我们有松散耦合的代码时，我们可以很容易地输入模拟或假的依赖关系，这样我们就可以很容易地分离出我们实际想要测试的代码部分。

**依赖注入的设计模式**

**构造器注入；属性注入；方法注入；环境上下文；服务定位器**

**实例研究**：

**非DI样本：**

示例应用程序将从一个存储库中获取数据（使用WCF服务）。对于表示层，我们将使用MVVM（模型-视图-视图模型）模式。

**代码层次**：

视图层

UI元素（XAML）

查看模型层

表示逻辑

存储库层

数据交互逻辑

服务层

实际数据访问

简单来看，示例程序似乎坚持了一些很好的面向对象的设计原则——比如单一责任原则(SinS。O.L.I.D.).**单一责任原则**规定，一个对象应该有一个（而且只有一个）改变的理由——这意味着它能做一件事，并且做得很好。

在我们的项目中，我们有一个独立的关注点——视图、视图模型、存储库和服务都在它们自己的单独的类中。这有助于实现可维护性，因为如果我们需要进行更改，我们知道在哪里进行更新。如果它与表示逻辑相关，则进入视图模型；如果它与数据交互相关，则它进入存储库。

有了这种关注点的分离，将使我们相信这段代码是松散耦合的。毕竟，一切都是分开的，都有自己的位置。但事实并非如此。**实际上，我们的代码是非常紧密耦合的。**下面根据代码进行分析来找到问题。

**该视图与该视图模型紧密耦合**

从我们的观点中考虑一下这段代码：

=新主模型（）；

因为视图的构造函数是“新的”主窗口视图模型的特定实例，所以它与该类紧密耦合。它对视图模型有一个直接的引用，并且我们不能构建我们的视图，除非在编译时也有主窗口视图模型的程序集。

**视图模型与存储库紧密耦合**

考虑一下我们的视图模型中的这段代码：

存储库=新的个人服务存储库（）；

因为视图模型的构造函数是“新的”建立的一个特定的实例人员服务存储库，它与该类紧密耦合。

**存储库与服务紧密耦合**

考虑一下来自我们的存储库中的这段代码：

服务代理=新的个人服务客户端（）；

因为存储库的构造函数是“新”的特定实例，所以它与该类紧密耦合。

**该视图与服务紧密耦合**

由于所有的紧密耦合，视图都与服务紧密耦合（通过视图模型和存储库）。

由于这里的应用程序比较小，所以这种紧密耦合可能看起来没有那么糟糕。毕竟，对于这样大小的应用程序，修改和跟踪正在发生的一切并不难。但我们只是把这个作为一个很容易理解的简单例子，而不会太陷入复杂的业务逻辑。但是当我们开始研究那些规模巨大的应用程序时，情况就会发生巨大的变化。

**紧密耦合的结果**

那么，我们为什么要关心这种紧密的耦合呢？下面是一些需要考虑的场景（我们将通过依赖注入来解决它们）。

**场景1：一个附加的存储库**

应用程序的要求也在不断变化。通过经验，我们可以了解更有可能改变的领域（取决于我们的商业环境）。对于这个示例，我可以很容易地想象，我们会想要为我们的客户端添加其他数据存储选项——可能通过保存到SQLServer（或其他数据库）或文本文件（CSV或XML）。

但是，我们将如何使用这段代码来处理一个额外的存储库呢？我们需要修改视图模型，以便它能够实例化个人服务存储库或个人服务存储库或个人csv存储库。每次添加新的存储库类型时，都需要再次修改视图模型。我们的目标应该是消除紧密耦合，这样我们就可以添加一个存储库，而不需要修改（甚至重新编译）我们的视图模型。

**场景2：高速缓存存储库**

对数据存储的呼叫是“昂贵的”（通过电线传输并与另一台服务器通信），特别是在移动设备上。如果我们能在客户端上缓存一些数据，这样，我们就不必在每次需要访问很少更改的数据时都进行网络呼叫了。

我们可以很容易地构建一个缓存存储库来处理这个问题，但是我们遇到了与上述相同的问题——我们需要修改我们的视图模型代码来接受这个新的存储库。此外，我们需要一些方法来使我们的缓存存储库灵活，以便它将与服务存储库或SQL存储库或CSV存储库一起工作。

**场景3：单元测试**

我们当前的解决方案中缺少的一件大事是单元测试。单元测试允许我们自动测试代码中孤立的部分。但是由于我们当前的代码是紧密耦合的，所以我们没有简单的方法来隔离这些部分以进行测试。（这不是不可能的，但重构代码比将隔离函数所需的测试包放在一起要容易得多）。我们需要能够测试我们的视图模型而没有存储库代码。我们需要能够测试我们的存储库代码而没有服务代码。

因此，简而言之，**我们的目标应该是添加一些“接缝”，允许我们隔离代码进行测试，而不引入我们现在所拥有的所有紧密耦合的依赖关系**。

**依赖注入可以帮助我们解决这些场景，使我们的代码易于扩展，提高可测试性，甚至包括不需要我们重新编译应用程序的运行时更改。**

下面来看看示例代码的另一个解决方案：依赖注入。我们将使用依赖注入来打破一些紧密耦合。

**将存储库注入到视图模型中**

良好的面向对象设计的原则之一是编程到一个接口，而不是一个具体的类型。我们可以通过添加一个存储库接口来打破视图模型和存储库之间的紧密耦合。这将在我们的两个组件之间添加一个抽象层。

**将视图模型注入到视图中**

将所需的参数添加到主窗口视图模型的构造函数中。

**组合依赖关系图**

实例化主窗口，然后显示它。

**缓存存储库**

上面，我们讨论了添加一个缓存存储库——这将保留数据的客户端副本，这样我们就不必在每次请求数据时调用服务器。另一个要求是，缓存存储库需要与我们想要的任何具体存储库一起工作——无论是个人服务存储库、个人csv存储库还是其他存储库。这使得它成为使用依赖注入的装饰模式的理想候选。

装饰品模式：

装饰者模式是一个非常简单的想法。它描述了一种包装现有类型、添加一些功能并公开与原始类型相同的接口的方法。这使得它成为现有类型的替代替换。

顺便说一句，这可以看作是遵循了Liskov替代原则（S.O.L.I.D.中的L）。从技术上讲，我们使用的是对象组合和接口，而不是直接继承，但100%兼容的对象替换符合一般原则。

使用缓存存储库组合对象：

为了使用缓存存储库，我们只需要更改我们的组合根目录。

**单元测试**

因此，**我们已经看到了依赖注入如何帮助我们使代码具有更强的可扩展性和可维护性。但它也可以使我们的代码更具可测试性。**

正如我们在第一个（非DI）样本中提到的，很难对视图模型进行单元测试，因为我们不能很容易地分离出各个部分。由于视图模型与存储库紧密耦合，因此该存储库也需要包含在测试中。如果存储库失败了，它可能会影响我们的测试，即使我们并没有尝试测试这部分代码。但由于我们现在使用DI来消除这种紧密耦合，我们有很好的“接缝”。这使得创建依赖项的模拟变得非常容易，这样我们就可以独立地集中精力测试视图模型代码。

**带有容器的依赖关系注入**

到目前为止，我们已经通过在组合根目录中手动连接依赖项来实现了依赖项注入。但是使用依赖注入最常见的方法是使用第三方依赖注入容器/框架。与构建我们自己的依赖注入容器相比，这些都提供了许多优势——包括终身管理、配置和依赖解析。有相当多的DI容器免费可用（而且许多也是开源的）。

**使用容器进行单元测试**

使用DI容器进行的单元测试与我们之前看到的单元测试非常相似。不同之处在于，在我们的设置方法中，我们将配置我们的DI容器；在我们的测试中，我们将从容器中解析对象，而不是自己实例化它们。

**总结**

我们已经看到了很多样本和很多DI的概念，以及如何使用依赖注入将紧密耦合的组件分解成孤立的、松散耦合的组件。我们已经使用DI来为代码添加接缝，以提高可测试性。我们已经看到了许多依赖注入模式，如构造函数注入和属性注入。我们已经研究了在组合根中连接依赖关系，以及如何使用DI容器来解析对象图中的依赖关系。我们也看到了依赖注射如何帮助我们坚持S.O.L.I.D原则这些良好的面向对象设计的原则，与其他设计原则一样，它们已经被证明在许多情况下可以实现更好的代码。我们没有提到的那个（“I”）是接口隔离原则。这个原则与我们如何将方法分割成不同的接口有关，这样类就只依赖于它们所使用的方法。

依赖注入是一组非常强大的模式。它并不一定适用于每种情况。但是，我们工具箱中的更多工具使我们成为具备更多用途的开发人员。