 在中小规模的软件中，对象和对象之间的协作关系就能够满足需要。但是当软件规模扩大，复杂度上升的时候，面向对象技术强调的协作却表现出另一个极端的特点－耦合度太高导致的复杂度。这时候就需要有一种新的方法来弥补面向对象技术的弱点。大规模软件的特点大规模软件主要特点是复杂度。比较典型的例子是集成性的项目。软件系统需要将各种各样的硬件、遗留系统、外部接口整合起来。其间可能遇到不同的硬件接口，不同的操作系统，不同的语言，不同的平台，不同的数据库，不同的消息中间件，不同的网络介质。这些都使得系统变得非常的复杂.面向对象技术的特点是通过对象之间的职责分工和高度协作来完成任务。这样的好处是代码量较少，系统布局合理，重用程度高。但是当对象的个数大量增加的时候，对象之间的高度耦合的关系将会使得系统变得复杂，难以理解。以前对于这个问题的方法是采用包（请参考拙作面向对象软件开发中对包的相关讨论）作为容器来组织对象，对象之间的依赖性将转化为包之间的依赖性。这种方法听起来有道理，但是在实际中仍会出现难以解决的问题。包仅仅只是容器。这意味着对对象的组织可以是任意的，而包之间依赖关系的设计则还是取决于对象的依赖。此外，包的设计和对象一样，缺乏一个统一的风格。而统一的风格正是大规模软件设计所必须的，因为这样可以有效改进系统的可理解性，这一点非常重要。面向组件编程面向组件编程的缩写

是COP。COP是对OOP的补充，帮助实现更加优秀的软件结构。组件的粒度可大可小，需要取决于具体的应用。在COP中有几个重要的概念：服务,服务（Service）是一组接口，供客户端程序使用。例如，验证和授权服务，任务调度服务。服务是系统中各个部件相互调用的接口；组件，组件（Component）实现了一组服务，此外，组件必须符合容器订立的规范，例如，初始化，配置、销毁。COP 是对一种组织代码的思路，尤其是服务和组件两个概念。在下文会提到Spring框架中，就采用了COP的思路，将系统看作一个个的组件，通过定义组件之间的协作关系（通过服务）来完成系统的构建。这样做的好处是能够隔离变化，合理的划分系统。而框架的意义就在于定义一个组织组件的方式。理解组件组件不是一个新的概念，Java中的javaBean规范和EJB规范都是典型的组件。组件的特点在于他定义了一种通用的处理方式。例如，JavaBean 拥有内视的特性，这样就可以通过工具来实现JavaBean的可视化。而EJB规范定义了企业服务中的一些特性，使得EJB容器能够为符合EJB规范的代码增添企业计算所需要的能力，例如事务、持久化、池等。所以，组件比起对象来的进步就在于通用的规范的引入。通用规范往往能够为组件添加新的能力（就像上面所讨论的），但也给组件添加了限制，例如你需要实现EJB的一些接口。以下我们将讨论组件的一些相关问题：组件的粒度组件的粒度是和系统的架构息息相关的。组件的粒度确定了，系统的架构也就确定了。在小规模的软件中，可能组件的粒度很小，仅相当于普通的对象，但是对于大规模的系统来说，一个组件可能包括几十，甚至上百个对象。因此，对使用COP技术的系统来说，

需要正确的定义组件的粒度。较好的定义粒度的方法是对核心流程进行分析。针对接口接口和实现分离是COP的基础，没有接口和实现的分离，就没有COP。接口的高度抽象特性使得各个组件能够被独立的抽取出来，而不影响到系统的其它部分。接口和实现分离有以下几个好处：1.在模块/组件/对象之间解耦。2.轻松的抽换实现，而不用修改客户端。3.用户只需要了解接口，而不需要了解实现细节。4.增加了重用的可能性。IOCIOC 是Inversion of Control的简称。它的原理是基于OO设计原则的好莱坞原则（The Hollywood Principle）：不要访问我，我们将访问你。也就是说，所有的组件都是被动的（Passive），所有的组件初始化和调用都由容器负责。IOC有几种实现的类型，包括基于方法参数调用的Method-based (M) IoC，它把组件传递给每个方法调用；基于接口的Interface-based (I) IoC（通常称为类型1），它使用接口来声明组件之间的依赖性，例如，Serviceable, Configurable；基于Setter方法的Setter-based (S) IoC（通常称为类型2），它使用setter方法来设置组件之间的依赖性；基于构造函数的Constructor-based (C) IoC（通常称为类型3）。Martin Fowler将IOC模式称为Dependency Injection模式。IOC是框架开发的一个基本原理。在开源软件中，不少的容器类框架都采用了IOC的思路。组件污染在IOC 的第一类型中，由于组件需要实现一些特定的接口，或是从某个类集成。这将使得组件受到一些约束（称为Invasive），例如组件移植不便。另一种情况是组件需要依赖于一个特定的容器，最为典型的就是EJB，组件无法脱离容器单独存在，这也使得组件受到约束。这两种情况都属于组件污染。最理想的组件是只专注于自

身工作的组件，它没有其它额外的逻辑。不过按照这种标准，目前大部分的代码都是不符合的。因此，目前在开源软件界出现了一些轻量级的容器框架，典型的如上文提到的PicoContainer和spring。他们的定位就是提供一个对组件管理的统一模式，而组件可以单独的使用，也可以放在另一个容器中，容器仅仅只是为组件提供了一些额外的功能，和组件本身没有直接的依赖。为什么我们说接口比继承好，很重要的一个原因就是接口更加灵活，组件的依赖性更弱，同样的，目前另一种做法则是采用一些标记性的语言来实现比接口更大的灵活度。例如基于xml的配置文件，以及J2SE1.5中将要引入的属性。组件的测试组件和容器的依赖脱离为组件测试提供了一个很好的环境。我们在测试一节中讨论过容器内测试往往是比较麻烦的，其原因就是在于上面所说的组件污染问题。例如在 spring框架中，组件是一个标准的JavaBean，你可以直接编写代码来设置组件的属性和定义组件之间的依赖关系（适用于自动化测试），也可以把这项工作交给spring容器（适用于开发和部署）。组件测试在测试的分类中属于集成测试。理解服务在讨论组件时我们谈论了组件粒度的问题。当组件的粒度不仅仅限于单个对象的时候。在构成组件的多个对象中，有些对象处于组件内部，不和其它的组件交互，有些对象需要和外部组件进行交互。后一种对象起的就是服务的作用。在设计模式中，这种设计被称为Facade模式。而在OO语言中，他们相当于接口的概念。不管如何比喻，服务订立了组件和组件之间的契约。这种契约是稳定的（如果业务需求是稳定的），不会随着组件内部的变化而发生变化。要理解这一点也非常的容易。对于一个提供用户认证的

组件，一个可能的服务对用户进行认证和授权，至于组件内部采用LDAP还是关系数据库来存放用户信息，对服务来说没有任何的差别。这样做的好处有很多，一是组件之间能够以一种稳定的方式存在，组件内部的变化不至于扩散到整个软件系统。二是软件设计将会转向重点设计组件之间的服务，而组件的实现细节将会隐藏起来，这不但有助于设计者更好的把握软件的全局架构，而且有助于分工的细化。服务并不是什么新颖的概念，RPC、IDL都是类似的技术。但我们谈的服务侧重架构和理念，不涉及到具体的技术，这一点同SOA和WebService的关系类似－SOA是一个结构性的概念，而WebService是实现SOA的一种适合的技术。 服务最好实现为接口。原则上服务可以是任何一种技术：JMS、WebService、RPC、或是简单方法调用。但是出于服务的稳定性的考虑，我们不应该将一个服务和具体的技术绑定起来，这样会使的服务发生变化的可能性增大。在Java语言中，接口是具有极大的灵活性的，因此，将接口实现为普通的Java接口是较好的选择。不过，这样做的话，我们也许就不能够使用远程调用，Web服务之类的功能了，不过这并不要紧，以下就是原因。服务适配器客户端可以直接使用接口，也可以通过适当的适配器来将普通的接口服务转换为特定技术实现的服务。如上图所示，一个普通的接口通过适配器模式转换成和特定技术相关的服务。在JMX技术中，也采用这种方式，JMX平台能够将一个普通的服务端口通过适配器进行转换，以适用于各种的协议，例如http、sock、snmp等等。 