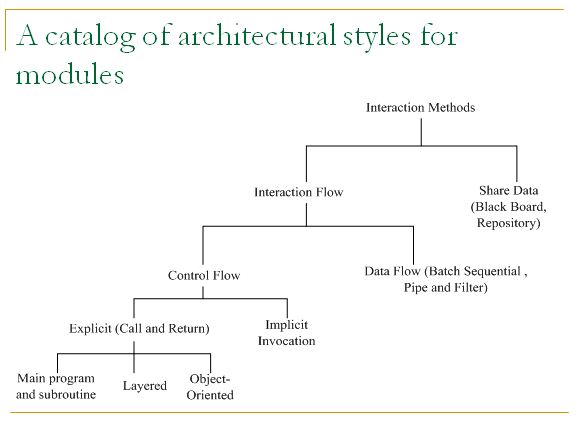
* 总体分类



# 主程序/子程序(MainProgram/SubroutineStyle)



子程序。主程序是系统的控制器，负责调度各子程序的执行。各子程序又是一个局部的控制器，调度其子子程序的执行。

主程序/子程序风格的重要设计决策与约束有：

基于声明-使用（程序调用）关系建立连接件，以层次分解的方式建立系统部件，共同组成层次结构。

每一个上层部件可以“使用”下层部件，但下层部件不能“使用”上层部件，即不允许逆方向调用。

系统应该是单线程执行。主程序部件拥有最初的执行控制权，并在“使用”中将控制权转移给下层子程序。

子程序只能够通过上层转移来获得控制权，可以在执行中将控制权转交给下层的子子程序，并在自身执行完成之后必须将控制权还交给上层部件。

主程序/子程序风格的优点有：

流程清晰，易于理解。严格的层次分解使得整个系统的结构组织非常符合功能分解和分而治之的思维方式，从而能够清晰地描述整个系统的执行流程，易于理解。

强控制性。严格的层次分解和严格的控制权转移使得主程序/子程序风格对程序的实际执行过程具备很强的控制能力，这带来了一个特点：如果一个子程序所连接的子子程序是正确的，那么就很容易保证该子程序的“正确性”。所以，主程序/子程序风格比其他常见风格更能控制程序的“正确性”

主程序/子程序风格的缺点有：

程序调用是一种强耦合的连接方式，非常依赖交互方的接口规格，这会使得系统难以修改和复用。

程序调用的连接方式限制了各部件之间的数据交互，可能会使得不同部件使用隐含的共享数据交流，产生不必要的公共耦合，进而破坏它的“正确性”控制能力。

# 面向对象式的(object-oriented)

面向对象式风格借鉴面向对象的思想，组织整个系统的高层结构。面向对象式风格将系统组织为多个独立的对象，每个对象封装其内部的数据，并基于数据对外提供服务。不同对象之间通过协作机制共同完成系统任务。

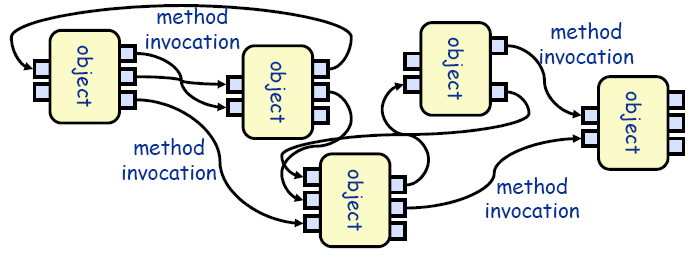
面向对象式风格的重要设计决策及约束有：

依照对数据的使用情况，用信息内聚的标准，为系统建立对象部件。每个对象部件基于内部数据提供对外服务接口，并隐藏内部数据的表示。

基于方法调用（Method Invocation）机制建立连接件，将对象部件连接起来。

每个对象负责维护其自身数据的一致性与完整性，并以此为基础对外提供“正确”的服务。

每个对象都是一个自治单位，不同对象之间是平级的，没有主次、从属、层次、分解等关系。



面向对象式风格适用于那些能够基于数据信息分解和组织的软件系统，这些系统：

主要问题是标识和保护相关的数据信息；

能够将数据信息和相关操作联系起来，进行封装。

实践中，基于抽象数据类型建立的软件系统大多属于面向对象式风格。

# 分层风格

分层风格根据不同的抽象层次，将系统组织为层次式结构。每个层次被建立为一个部件，不同部件之间通常用程序调用方式进行连接，因此连接件被建立为程序调用机制。

分层风格的重要设计决策与约束有：

从最底层到最高层，部件的抽象层次逐渐提升。每个下层为邻接上层提供服务，每个上层将邻接下层作为基础设施使用。也就是说，在程序调用机制中上层调用下层。

两个层次之间的连接要遵守特定的交互协议，该交互协议应该是成熟、稳定和标准化的。也就是说，只要遵守交互协议，不同部件实例之间是可以互相替换的。

跨层次的连接是禁止的，不允许第I层直接调用I+N（N>1）层的服务。

逆向的连接是禁止的，不允许第I层调用第J（J<I）层的服务。

分层风格的优点有：

设计机制清晰，易于理解。通过将系统按照不同的抽象层次组织为层次结构，分层风格可以将混杂的耦合逻辑分解为几个不同的部分（例如网络通讯协议的分层），每个部分变得更简单、纯粹和易于理解，从而使得整个设计机制非常清晰。

支持并行开发。分层风格的不同层次之间遵守成熟、稳定和标准化的交互协议，也就是说一旦层次之间的连接明确下来，就很少会发生改变。而且只要不破坏交互协议，每个层次内部的开发决策不会对其他层次内部的开发决策产生影响。所以分层风格能够支持并行开发，它的每个层次都可以交给一个团队进行独立开发。

更好的可复用性与内部可修改性。因为不同层次之间通过成熟、稳定的交互协议通信，因此，只要遵守其交互协议，不同的层次部件就能够互相替换，具有很好的可复用性。在不影响交互协议的情况下，每个层次可以自由安排其内部实现机制，因此分层风格也具有很好的内部可修改性。

分层风格的缺点有：

交互协议难以修改。虽然分层风格能够实现很好的内部可修改性，但是它难以修改交互协议，这也是它要求交互协议比较成熟和稳定的原因。因为，一方面，对交互协议的修改意味着层次的对外行为需要变更；另一方面，不同层次是对同一系统的不同程度抽象，因此对外行为常常是存在于所有层次，只是抽象程度不同而已。最后，如果交互协议需要改变，那么就可能需要改变所有的层次。

性能损失。分层风格禁止跨层调用，这使得每一个外界请求都需要沿着层次逐一深入，多次调用，这可能会产生冗余的调用处理，带来不必要的性能损失。

难以确定层次数量和粒度。如果层的粒度太大，就只会有少数几个层次，不能完全发挥分层风格的可复用性和内部可修改性。反之，如果层的粒度太小，层次的数量就会太多，引入不必要的复杂性，带来额外的性能损失。

分层风格适用于具备下列特性的系统：

主要功能是能够在不同抽象层次上进行任务分解的复杂处理；

能够建立不同抽象层次之间的稳定交互协议；

没有很高的实时性能要求，能够容忍稍许的延迟；

此外，那些需要进行并行开发的软件系统也可能会使用分层风格，以便于任务分配和工作开展。

# 隐式调用风格

隐式调用风格（Implicit Invocation Style），又被称为发布-订阅风格（Publish Subscribe Style）、基于事件的风格（Event Based Style）和选择性广播风格（Selective Broadcast Style）

隐式调用风格将系统中的不同功能部分建立为部件，并使用事件（而不是程序调用）来实现不同部件之间的连接，也就是说隐式调用风格使用事件传播机制——事件路由，来建立连接件。

隐式调用风格中的行为者仍然会声明可调用程序来提供对外的服务，但是这些程序却不会直接被外界调用。隐式调用风格采用的机制是将特定的事件类型与行为者的程序联系起来，在相应事件发生后，通过联系可以找到需要调用的程序并将其调用执行，这就是该风格被称为“隐式调用”的原因。对可调用程序接口的管理及其与事件类型的匹配都是连接件的工作。

隐式调用风格的重要设计决策与约束有：

如果部件A和部件B需要相互协作，实现A调用B的效果，那么隐式调用风格的协作机制为：○1部件A向部件路由R声明它将要抛出的事件类型e；○2部件B在事件路由R中注册“事件-程序”的映射关系<e,p>，表明如果发生事件e，事件路由R可以调用B的程序p；○3在部件A需要与部件B协作时，部件A向事件路由广播事件e；○4事件路由根据注册的映射关系，将部件B的程序p调入执行。

多个部件可以声明同一个事件类型，每一个部件的事件广播都有相同的调用效果。

多个部件可以注册同一个事件类型，并在事件发生后同时被调用执行；

事件的广播者不知道哪些部件会受到影响，不能假设事件对部件的影响关系，甚至不能假设事件一定会有接收者。

部件不能假设对事件的处理顺序，也不能假设对事件的处理结果。

隐式调用风格的优点有：

可复用性。只需要注册一个事件，任何一个部件都可以融入系统，不再存在接口兼容与匹配的问题，所以隐式调用风格的一个优点是能够很好地支持可复用性。

可修改性。在隐式调用风格中，部件实例对交互的参与只依赖于事件类型，不受接口规格的影响，所以只需保持对事件类型的依赖不变，就能够完成部件实例内部和接口的变更。即使要修改对事件类型的依赖，也相对比较容易，因为它对事件类型的依赖关系是动态维护而非静态绑定的。

性能。如果隐式调用风格的部件实例采用进程实现，就可以做到多进程并发执行，所以隐式调用风格有潜在的高性能优点。

隐式调用风格的缺点有：

弱控制性。在隐式调用风格中，任何一个部件都无法控制整个系统的执行流程，当发布一个事件时，甚至不能保证会有其他部件做出反应，对于有反应的部件，也不能假设其处理结果。所以，隐式调用风格在保证程序“正确性”上有更多的困难，具有弱控制性。

难以测试和验证。在程序调用机制中，只需要保证程序执行前和执行后的一致性，就能够测试和验证程序的“正确性”。但是在隐式调用风格中，广播事件和接收事件的部件双方都不知晓对方，也不知晓接收事件的处理顺序，所以它们无法确认被调用程序执行前的上下文环境，不能保证程序执行前的一致性，也就难以测试和验证程序的“正确性”。

隐式调用风格适用于能够以松散耦合部件为基础建立的软件系统，这些系统中的每个部件都执行一些操作，并可能会顺次调用其他部件的操作。

在实际应用中，隐式调用被看作是一种重要的程序设计技术和集成手段，得到了广泛的应用。例如，编译环境中的工具集成，数据库管理系统中的一致性检查，图形化用户界面等，都有很多应用隐式调用风格的成功案例。

# 管道-过滤器风格：

管道-过滤器风格将系统的功能逻辑建立为部件集合。每个部件实例完成一个对数据流的独立功能处理，它接收数据流输入，进行转换和增量后进行数据流输出。连接件是管道机制，它将前一个过滤器的数据流输出传递给后一个过滤器作为数据流输入。连接件也可能会进行数据流的功能处理，进行转换或增量，但连接件进行功能处理的目的为了适配前一个过滤器的输出和后一个过滤器的输入，而不是为了直接承载软件系统的需求。

管道-过滤器风格最重要的设计决策与约束是保证过滤器的独立性，即：

每个过滤器都独立工作，无需知道通过管道与其相连的其他过滤器的特征。也就是说，每个过滤器只需要了解输入数据流和保证输出数据流即可，不用关心一起工作的其他过滤器的实现细节。

过滤器之间不能共享任何状态，更不能共享数据。

虽然在软件体系结构设计时需要对各过滤器的执行顺序进行专门安排，但是整个过滤网络的正确输出不能依赖于各过滤器的执行顺序。也就是说，从系统设计的角度讲，需要专门安排各过滤器的执行顺序，但是在实现单个过滤器时，不能将结果的正确性依赖于执行顺序上。

各个过滤器可以并发执行。每个过滤器都可以在数据输入不完备的情况下就开始进行处理，每次接到一部分数据流输入就处理和产生一部分输出。这样，整个的过滤器网络就形成了一条流水线。

管道-过滤器风格的优点有：

可复用性。因为管道-过滤器风格要求过滤器具有独立性，所以任何两个过滤器都是互不直接影响的。只要输入和输出的数据流内容符合要求，就可以通过建立相应的管道将一个过滤器融入一个软件系统，所以管道-过滤器风格具有很好的可复用性。

内部可修改性。只要不修改输入和输出的数据流，过滤器可以自由安排对内部实现的修改，所以管道-过滤器风格具有较好的可修改性。

可扩展性。因为过滤器的正确执行不依赖于前面与后面的过滤器，也不依赖于整个过滤器网络的执行顺序，所以在整个过滤器网络中，只要在某一环节添加一个新的过滤器，就能够扩展系统的功能，也就说管道-过滤器风格有很好的可扩展性。

性能。管道-过滤器风格使用了并发执行的流水线机制，所以具有很好的性能。

支持特定分析。管道-过滤器风格还支持某些特定的分析，例如吞吐量和死锁检测。

管道-过滤器风格的缺点有：

弱控制性和弱交互性。管道-过滤器风格的过滤器是独立和并发执行的，而且它们对数据流的处理采用了流水线的方式，所以各个过滤器是各自为政的，系统很难找到一种方式来不间断控制全系统的数据处理情况，也很难提供一种能够统揽全局持续变化的交互方式。

空间效率较差。因为过滤器之间不能共享数据，所以即使使用同样的数据，每个过滤器也都要保留自己的数据副本，导致空间效率较差。

性能浪费。管道的数据传输会耗费一定的性能，尤其是如果管道需要在传递数据的同时对其进行转换与处理，那么额外的性能浪费就更加严重。

错误处理能力较弱。因为过滤器之间相对独立，所以，如果一个过滤器发生了错误，那么其他的过滤器并不能及时感知并纠正错误。即使发生错误的过滤器及时发出了警报，也会有部分已经发生了错误的数据会传递出去，只有重启整个过滤器网络才能完全恢复对所有数据的正常处理。所以，如果软件系统不能容忍任何错误，或者不允许重新启动，那么错误能力较弱的问题就会成为管道-过滤器风格的致命缺点[Buschmann2002]。

# 存储库风格

存储库风格将系统的功能处理建立为一系列的知识源部件，它们收集和处理系统的数据信息，完成系统的功能任务。存储库风格还建立了一个共享数据部件，它储存系统所有的数据信息，代表了系统的状态。知识源部件并不储存数据，所以在它们进行数据收集与处理时，需要访问共享数据区，这是通过直接进行存储区访问实现的，存储库风格将对存储区的直接访问建立为连接件。

存储库风格的重要设计决策与约束有：

所有的知识源相互独立，它们彼此不互相调用，它们的活动也没有预先确定顺序。

所有的知识源都依赖于共享数据，不仅它们处理的数据来自于共享数据，而且它们的执行流程和顺序也要取决于共享数据的状态。

知识源负责实时检查共享数据的状态，并在必要时做出合理的反应。

存储库风格的优点有：

空间效率。因为所有的知识源交互都需要经过共享数据来传递，所以存储库风格要求共享系统的全部重要数据信息，这无疑使得它具有很好的空间效率。

性能。在进程实现时，存储库风格允许多进程并发，所以它具有潜在的性能优势。

知识源的可修改性。存储库风格要求各个知识源是相互独立的，只需要依赖于共享数据。因此，只要不影响共享数据，各个知识源可以很容易地进行修改，具有较好的可修改性。

容错性和健壮性。存储库风格将全局的数据都存储在一起，这使得它可以集中精力保障系统的容错性和健壮性，例如建立共享数据的备份、控制共享数据的安全、控制共享数据的并发等。而且，与拥有部分数据相比，拥有所有的数据能够更准确地判断模糊与不确定问题，这也能提高系统的容错性和健壮性。

存储库风格的缺点有：

共享数据的难修改性。因为所有的知识源都要依赖于共享数据，所以共享数据一经建立，就难以修改，其维护的代价非常高昂。

共享数据的瓶颈性。共享数据在存储库风格中起着举足轻重的作用，如果共享数据

发生了性能低下、数据丢失、安全隐患等问题，那么整个系统都将受到影响。

弱控制性。存储库风格用共享数据实现交互的方式非常类似于公共耦合，知识源之间的影响范围会隐形扩大，而且难以预期和不易察觉。知识源的行为还需要受到共享数据状态的影响，这使得一些行为一旦执行就很难再现和测试。所以，存储库风格具有弱控制性。

# 模型-视图-控制风格(MVC)

模型-视图-控制风格以程序调用为连接件，将系统功能组织为模型、视图和控制三种连接件。模型封装了系统的数据和状态信息，实现业务逻辑，对外提供数据服务和执行业务逻辑。视图封装了用户交互，提供业务展现，接收用户行为。控制封装了系统的控制逻辑，根据用户行为调用需要执行的业务逻辑和数据更新，并且根据执行后的系统状态决定后续的业务展现。

模型-视图-控制风格的重要设计决策和约束有：

模型、视图、控制是分别是关于业务逻辑、表现和控制的三种不同内容抽象。

如果视图需要持续地显示某个数据的状态，那么它首先需要在模型中注册对该数据的兴趣。如果该数据状态发生了变更，模型会主动通知视图，然后再由视图查询数据的更新情况。

视图只能使用模型的数据查询服务，只有控制部件可以调用可能修改模型状态的程序。

用户行为虽然由视图发起，但是必须转交给控制部件处理。对接收到的用户行为，控制部件可能会执行两种处理中的一种或两种：调用模型的服务，执行业务逻辑；提供下一个业务展现。

模型部件相对独立，既不依赖于视图，也不依赖于控制。虽然模型与视图之间存在一个“通知变更”的连接，但该连接的交互协议是非常稳定的，可以认为是非常弱的依赖。

模型-视图-控制风格的优点有：

易开发性。模型、视图、控制是分别是关于业务逻辑、表现和控制的三种不同内容抽象，设计机制清晰，易于开发。

视图和控制的可修改性。模型封装了系统的业务逻辑，所以是三种类型中最为复杂的系统部件。MVC中模型是相对独立的，所以对视图实现和控制实现的修改不会影响到模型实现。再考虑到业务逻辑通常比业务表现和控制逻辑更加稳定，所以MVC具有一定的可修改性优势。

适宜于网络系统开发的特征。MVC不仅允许视图和控制的可修改性，而且其对业务逻辑、表现和控制的分离使得一个模型可以同时建立并保持多个视图，这非常适用于网络系统开发。

模型-视图-控制风格的缺点有：

复杂性。MVC将用户任务分解成了表现、控制和模型三个部分，这增加了系统的复杂性，不利于理解任务实现。

模型修改困难。视图和控制都要依赖于模型，因此，模型难以修改。

应用：因为有适宜于网络系统开发的特征，所以MVC风格主要用于网络系统的开发。

# 客户端/服务器风格（CS）：

客户端/服务器风格将系统的功能进行了划分。划分后的一部分功能被建立为服务器部件，它们通常有较高的资源要求。系统中另一部分功能被建立为客户端部件，它们需要处理更多的用户交互。网络连接被建立为连接件，它将客户端的请求发送到服务器，服务器提供服务满足请求，网络连接再将服务的结果返还回客户端。

客户端/服务器风格的重要设计决策与约束有：

服务器较为固定，每个客户端都知道服务器的标识；

客户端可以动态增减，服务器不知道客户端的标识；

各个客户端之间相互独立，它们都依赖于服务器。

客户端/服务器风格的优点有：

易开发。在网络软件系统中，网络通信复杂度是开发的难点之一。客户端/服务器风格限定了Server的标识，这使得网络通信功能的设计和开发变得容易。

客户端的动态性。客户端/服务器风格中的客户端可以随时增减，能够实现客户端的动态性，这非常符合实际应用的特点。

客户端/服务器风格的缺点有：

服务器难以调整。因为所有的客户端都要知道服务器的标识，都要依赖于服务器，所以服务器难以调整。

服务器瓶颈。计算复杂性较高的服务通常都由服务器承载，这可能会造成服务器负载超重，成为瓶颈。

不易更新。因为客户端和服务器端都有实现，要部署相应的程序代码，所以，如果系统发生了修改，那么在更新程序部署时，就必须要更新所有的客户端和服务器端，这个工作量还是比较繁杂的。