**chapter 模块化**

* **理解模块化**

（结构化）模块化的设计思想使得我们能够从简单的、独立的、可复用的模块开始，构建复杂的系统，从而达到复杂系统的**易开发、易修改、易复用**。

* **模块化的优点**
* 从简单性考虑：

减少了调试和修改的时间。

通过将系统划分为独立的各个模块，每个模块能够在极少的去考虑对其他模块的影响下，独立的被考虑、实现、修改。

* 从可观察性考虑：

每个模块在逻辑上是清楚正确的，能够很容易的看出逻辑行为是如何以及为何发生。

减少了开发和修改的时间。

* 从模块的角度考虑：

系统的其他部分可以通过模块的名字对该模块进行调用。

每个模块只专注于解决一个问题，减少了问题的复杂度，易于实现和修改。

在模块化的思想下，每个模块能够得到**独立**的对待：

独立理解

独立使用或复用

独立被构建

一个模块的失效不会影响到其他模块

一个模块的修改不会影响到其他模块

* **降低耦合：最小化模块之间的联系**

**耦合**：模块之间建立的联系的强度的度量

耦合分为两大类：模块与公共环境之间的联系、模块与模块之间的联系。

**->原则1：共享变量是有害的！**

参照公共耦合。

**->原则2：语法语义清晰，降低隐式关系**

理解一个模块不需要依赖其他模块的含义。（PPT例子）

**->原则3：避免重复**

否则，外部的修改会导致各个模块中重复的部分也需要修改。

**->原则4：面向接口编程**

接口：模块自身

实现：模块内部

通过模块的名字与整个模块建立的联系产生的耦合 < 与模块内部元素产生的耦合

* **内容耦合：模块之间的完全依赖**

数据和控制元素混合在一起，通过一个模块来修改另外一个模块，被修改的模块完全依赖与修改它的模块。

一个模块完全依赖与另外一个模块的内容。（PPT例子：goto）

* **公共耦合：共享全局变量、拥有共享的上下文环境**

解决方案：

提高信息内聚，实现数据封装，减少公共耦合。

将共享数据进行子划分。

将共享数据的访问限制于最小的子模块。

* **控制耦合**

传递必要的参数以外的信息，而且参数的内容影响模块的逻辑，影响**程序执行的分支**。经常出现的现象：接收到参数，然后开始进行if-else或switch判断。

解决方案：将判断转移到传送参数的模块，使得耦合关系变成可以接受的数据耦合。

* **印记耦合**

传递必要的参数以外的信息。例如，传递的参数为某一复杂的数据结构，依赖与该数据结构的结构信息。

* **数据耦合**

传递必要的参数产生的联系。（PPT例子）

**内容耦合 > 公共耦合 > 控制耦合 = 印记耦合 > 数据耦合**

内容耦合最差

公共耦合也不可接受

数据耦合好一些

* **提高内聚：最大化模块内部元素的关系**
* **偶然内聚**

随意放在一起。main函数？

* **逻辑内聚**

在**逻辑上相似（加减乘除）**。

if-else switch-case

EDIT ALL DATA

EDIT: PROC (TYPE, DATA)  
/\* do some common edits \*/  
IF TYPE = “A”  
THEN /\* edit by type A rules \*/  
IF TYPE = “B”  
THEN /\* edit by type B rules \*/

* **时间内聚**

一系列的事务在同一时间段内完成。

例如：所有的**初始化**

* **通信内聚**

由于**对共同数据的引用**而建立的内聚。

如果一个模块的所有成分都操作同一数据集或生成同一数据集，则称为通信内聚。

“Read” and “Print” input data

* **顺序内聚**

为解决某个问题而实现一系列**顺序的步骤**。

如果一个模块的各个成分和同一个功能密切相关，而且一个成分的输出作为另一个成分的输入，则称为顺序内聚。

read deal update output

* **功能内聚**

具有一个共同的目标，完成一个功能。

* **信息内聚**

**信息内聚 > 功能内聚 > 顺序内聚 = 通信内聚 = 时间内聚 > 逻辑内聚 > 偶然内聚**

* **【面向对象】中的耦合关系**

1. **交互耦合**

同结构化思想中的耦合关系相同，包括：

共享数据

变量访问

方法调用

隐式关系

1. **组件耦合：类之间的关系，一个类使用另一个类的实例对象（PPT例子）**

组件耦合的四种类型：

全局的变量：聚合关系

参数：方法调用之间的传参

创建：方法内部的局部变量

隐式关系：由其他对象供给

组件耦合又可分为以下三种类型：

1. 隐式组件耦合：（PPT）例子

最差的耦合关系。

C’的对象出现在C的某个方法实现中，但是C’在C的规格说明和实现中都没有说明。

1. 分散的组件耦合：

C’作为局部变量或实例变量在C的实现中出现，但是没有在C的规格中说明。

表现：

聚合关系：全局变量

局部变量

分散式组件耦合是可以接受的。

1. 规格化组件耦合：

C’在C的规格中说明。

1. **继承耦合**
2. **修改式**继承耦合：（P+C=1+N）

最差的继承耦合。

子类对父类进行没有任何规则的修改。

如果外界拥有父类的引用，必须知道父类+子类所有方法。

1. **改进式**继承耦合：（P+C重写的一些方法=1+小N）

在预定义的基础上的修改。

如果外界拥有父类的引用，必须知道父类+子类修改的。

1. **扩展式**继承耦合：（ONLY P=1）

继承的都没有修改，只是添加方法和变量。

如果外界拥有父类的引用，只需要知道父类。

**chapter 信息隐藏**

* **信息隐藏的含义**

每个模块都隐藏了关于一个设计决策的实现。

所有的设计决策都是相对独立的。

* **KWIC的两种模块化划分方法**

第一种模块化划分方法：

按照【处理流程】的方法进行划分，处理过程中每个主要的处理步骤封装成一个模块。

必须在并行工作之前设计好所有的数据结构，并需要复杂的描述。

第二种模块化划分方法：

按照【信息隐藏】的思想进行划分，每个模块封装一个或几个“秘密”，每个模块都隐藏了一个独立于其他模块的设计决策。

必须在并行工作之前设计好所有的接口，只需要简单的描述。

两种模块化划分的结果，可能都共享相同的数据结构和相同的算法，不同的是职责分配时模块划分的方式。系统运行是的表现可能是相同的，但是在可修改性、文档化、可理解性等方面的展现不同。

**->设计定律1**：不要按照处理流程中的步骤分解模块。

* **Module Guide**

针对复杂的系统

**->设计定律2**：创建一个层次化结构的文档——module guide（自顶向下的居多，也可以改成自底向上的）

按照module guide的思想，每个模块包括以下3个部分：

1. **秘密**：主要秘密和次要秘密
2. 主要秘密（来自SRS软件需求规格）

软件设计时希望隐藏的信息。

1. 次要秘密（来自变化）

实现隐藏主要秘密的模块的设计时的实现决策。

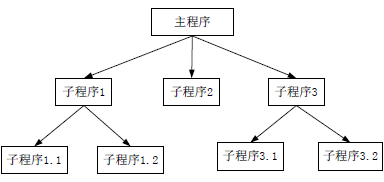
1. 模块在系统运作用扮演的**角色**（功能）
2. 模块提供的**接口**（？面向接口编程）

**chapter 体系结构设计风格【部件】【连接件】【约束】**

* **模块层次的设计风格**

调用—返回风格（主程序—子程序风格、面向对象风格）

1. **主程序—子程序风格**

****

1. 描述

主程序/子程序风格将系统组织成**层次结构**，包括一个主程序和一系列子程序。**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：

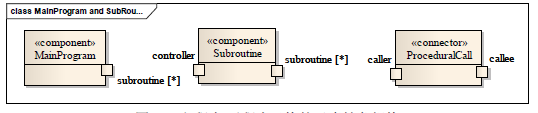
基于程序调用关系建立连接件，以层次分解的方式建立系统部件，共同组成层次结构。

不允许逆方向调用：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。

 单线程执行：主程序部件拥有最初的执行控制权，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。

1. 抽象规格



【主程序部件】：它的实例是整个系统的控制器，控制权的发起者和终结者。subroutine端口包含需求接口，定义了对子程序的功能与交互期望。

【子程序部件】：处于主程序下层，controller端口描述了对上层部件提供的服务。

【程序调用连接件】：程序调用机制。caller定义了给调用者提供的交互机制，callee定义了给被调用者提供的交互机制。

1. **面向对象式风格**
2. 描述

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：

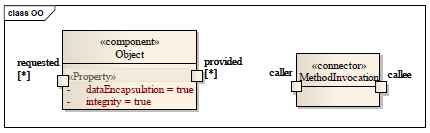
依照对数据的使用情况，用信息内聚的标准，为系统建立对象部件。每个对象部件基于内部数据提供对外服务接口，并隐藏内部数据的表示。

基于**方法调用**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。

1. 抽象规格



【对象部件】：自治的对象。

【方法调用连接件】：典型的方法调用机制。

1. **管道——过滤器风格**
2. 描述

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：

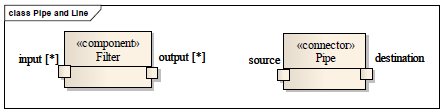
每个过滤器都独立工作，无需知道通过管道与其相连的其他过滤器的特征。也就是说，每个过滤器只需要了解输入数据流和保证输出数据流即可，不用关心一起工作的其他过滤器的实现细节。

过滤器之间不能共享任何状态，更不能共享数据。

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。

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。

1. 抽象规格



【过滤器部件】：对数据流进行转换或增量的功能处理部件。

【管道连接件】：传递数据流的连接件。

1. **隐式调用风格**
2. 描述

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。

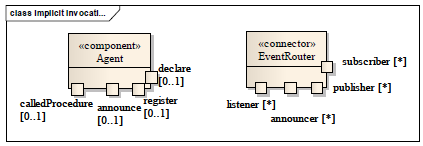
多个部件可以声明同一个事件类型，每一个部件的事件广播都有相同的调用效果。

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。

1. 抽象规格



【Agent部件】：需要利用隐式调用机制实现协作的部件。declaration端口含有需求接口，描述了对事件声明服务的期望，事件生命服务允许部件声明自己将要广播的事件类型。register端口含有需求接口，描述了对事件注册服务的期望，时间注册服务允许部件注册自己感兴趣的事件类型。端口announce含有需求接口，描述了对事件广播服务的期望，事件广播服务允许部件将一个事件广播出去。端口calledProcedure含有供给接口，描述了相关联事件发生后将会执行的部件服务。

【事件路由连接件】：角色publisher描述了EventRouter为事件声明者提供的交互协议。角色subscriber描述了EventRouter为事件注册者提供的交互协议。角色announcer描述了EventRouter为事件广播者提供的交互协议。角色listener描述了EventRouter对事件接收者的行为期望。

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。

1. **存储库风格**
2. 描述

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。

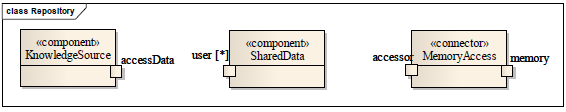
存储库风格的重要设计决策与约束有：

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。

1. 抽象规格



【知识源部件】：收集和处理系统信息，完成系统功能任务的部件。端口accessData包含需求接口，描述了它期望外界所能提供的数据访问交互。

【共享数据部件】：储存系统所有数据信息的部件。端口user包含供给接口，定义了它对外提供的数据访问交互。

【内存访问连接件】：典型的存储器访问机制。角色accessor定义了给访问者提供的交互行为。角色memory定义了对存储器的交互行为期望。

人们经常将交互分为“拉（pull）”和“推（push）”两种模式，它们都是站在交互发起者的立场来描述的。“拉”（类似于拉的动作）是指交互发起者主动从外界查询和获得信息给自己，然后自行决定下一步行动。而“推”（类似于推的动作）则是指交互发起者将自己的信息告知给外界，由外界决定下一步行动。

存储库风格采用的是“拉”模式，由知识源主动发起**查询**，并根据共享数据状态决定下一步行动。

如果将知识源与共享数据的交互变为“推（push）”模式，由共享数据监控状态的变化，并在需要时通知知识源，然后再由知识源决定下一步行动，那么这种变体形式就是黑板风格（Blackboard Style）。

因为要监控状态变化，所以黑板风格需要强化共享数据部件的控制功能，或者在共享数据部件之外，再建立一个控制部件。

假设黑板风格除共享数据部件之外又建立了一个控制部件，那么它就拥有了三种部件类型（知识源、共享数据、控制）和3种连接件类型（数据访问：连接知识源与共享数据；监控：连接共享数据与控制；事件通知：连接控制与数据源），其交互机制为：

（1） 知识源部件A首先在控制部件中注册感兴趣的数据及其状态条件；

（2） 控制部件持续监控被注册了兴趣的数据及其状态条件；

（3） 另一个知识源部件B对共享数据的修改使得被注册了兴趣的数据满足了状态条件；

（4） 控制部件广播事件，通知所有注册了相应数据及其状态条件的知识源；

（5） 知识源A接收到事件，进行功能处理。

因为黑板风格对共享数据进行了全局性的控制，所以它能够更好地解决模糊性与不确定性问题。

实际应用中，专家系统、集成开发环境、聊天室等都采用了黑板风格。

存储库风格与黑板风格的比较

存储库风格使用“拉”模型：

组件向存储库中写数据

组件从存储库中读数据

容易实现

客户端复杂，且必须回滚数据

黑板风格使用“推”模型：

组件注册订阅数据

数据可获取是组件被通知

客户端程序简单

需要复杂的基础

1. **分层风格**
2. 描述

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：

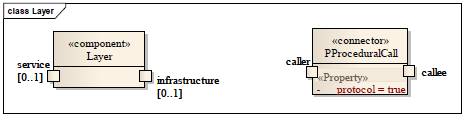
从最底层到最高层，**部件的抽象层次逐渐提升**。**每个下层为邻接上层提供服务，每个上层将邻接下层作为基础设施使用**。也就是说，**在程序调用机制中上层调用下层**。

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。

1. 抽象规格



【层次部件】：service端口为上层部件提供服务，包含供给接口，infrastructure端口对下层部件提供服务期望，包含需求接口。

【程序调用】：遵守特定交互协议的程序调用机制。支持并发。

* **进程层次的设计风格**

1. **点对点风格**

分布式系统中的同步消息传递机制。

松耦合，组件相互独立。

1. **发布—订阅风格**

多个应用希望接收相同的消息（广播）。

发布者和订阅者是解耦的

消息传递是基于事件订阅的

* **物理单元层次的设计风格**

1. **客户端—服务器风格**
2. 描述

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。

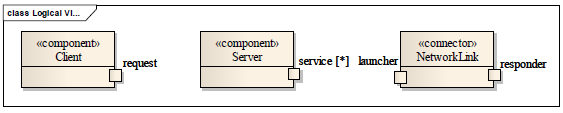
客户端/服务器风格的重要设计决策与约束有：

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。

1. 抽象规格



【客户端部件】：主要承载用户交互功能的部件。

【服务器部件】：承载复杂功能的部件。

【网络连接连接件】：典型的网络连接机制。

1. 变化形式
2. 三层风格

依据客户端和服务器分担职责的不同，客户端/服务器风格可以分为“瘦（Thin）”客户端和“胖（Fat）”客户端两种类型。

“瘦（Thin）”客户端类型将更多的功能部署到服务器上，减轻客户端负担的同时，也加重了服务器的负载，使得服务器的瓶颈缺点越发明显。B/S风格就是典型的“瘦（Thin）”客户端类型。

“胖”客户端类型将更多的功能部署到客户端上，以减轻服务器的压力。但是客户端功能过多又会增加系统的网络通信复杂度，使得系统开发更加困难，同时修改之后的程序更新工作也会更加繁杂。

为了解决“瘦”与“胖”之间的两难选择，人们提出了一种客户端/服务器风格的变体形式三层（3-Tier）。

三层风格在客户端与服务器之间加入一个新的层次——业务逻辑层，如图

三层风格分为三个层次：

**表现层**类似于C/S的瘦客户端，主要负责业务表现和用户交互，部署在网络中众多用户使用的物理单元上。

**业务逻辑层**负责系统的功能逻辑，完成系统任务，部署在网络中的业务逻辑服务器节点。表现层与业务逻辑层是C/S关系，表现层为Client，业务逻辑层为Server。

**数据服务层**负责系统的数据服务，部署在网络中的数据服务器节点。业务逻辑层与数据服务层是C/S关系，业务逻辑层为Client，数据服务层为Server。

按照同样的道理，业务逻辑层也可以进行分解，分为多层部署，以减轻每一层的服务器负载，此时该变体称为N层（N-Tier）风格。

最后需要强调的是，这里的“层”与分层风格的“层”是不同的概念，分层风格的“层”英文为Layer，是模块的组织，三层的“层”英文为“Tier”，是网络部署的组织。

1. 端到端风格

前面所述的C/S风格及其变体形式都明确区分了客户端和服务器，主要的功能都由服务器来承担。

但是，有些系统的功能负载非常高，很难由一个或几个服务器来承担。因此，又产生了一种类似于分布式系统风格的端到端（Peer to Peer）风格。

在端到端风格中，每个参与者都既是客户端，又是服务器

1. 分布式系统

**chapter 理解软件体系结构**

软件体系结构的三个方面：高层次结构、关注点、设计决策

* **详细设计机制的不足**

1. 关注点偏差（载体失配）
2. 作为载体，程序语言可以描述数据结构+算法，以及由此衍生的模块、类等结构，能够描述功能逻辑，但无法描述可靠性、性能等整体结构的质量要求
3. 根据最初的设计，程序设计语言不承载质量要求
4. 无法实现交互信息本地化**（**信息隐藏局限性）
5. 单词匹配的需求使得模块间必须依赖于对方的接口，导致信息隐藏的局限性
6. 模块间复杂连接逻辑被分散到其内部类之间，既突破了”NO INSIDE”的限制，又使得理解变得困难
7. 无法有效抽象部件的整体特性（无法描述module自身）
8. 总是进行模块内部的描述
9. 接口联合起来能够表现module的功能，但无法表现一个module的质量特性
10. 接口的定义缺乏结构性
11. 模块的众多接口之间太独立
12. 无法实现区别对待：为不同的其他模块公开不同的接口，尤其是不同的质量要求
13. 不能有效适应大型软件的特殊开发方法
14. 复用与单次匹配
15. 多语言/多范型与联合编译
16. 遗留资产与module inside使用

* **高层次结构的特点**

部件+连接件+配置

模块本身包含质量属性（部件）

模块之间的交互由一个独立的组件完成（连接件）

部件和连接件通过配置进行关联

* **高层次结构——理解软件体系结构的三个方面之一**

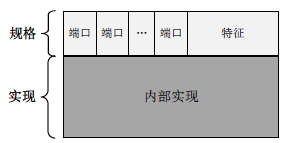


部件关注处理和数据；

连接件关注部件之间的交互。

连接件单独作为一个组件被考虑的原因：

* + The definition of a connector should be localized
  + 连接件可能非常的复杂。
  + 系统中的交互信息很难放在某个部件中
  + 所有的元素都应该被独立出来



* **【部件】**

软件体系结构中**封装处理和数据**的组件。

部件包括抽象规格与具体实现两个部分。

抽象规格：包括端口和特征集。

特征集：包括部件的类型、功能性、约束、质量属性等特征。

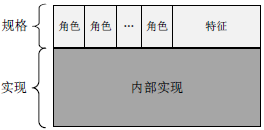
端口：对外可见、可被外界引用的接口实体，代表了部件对外承诺的一种职责。

部件可分为两种类型：原始部件和复合部件。

原始部件可以直接被实现为响应的软件实现机制。

|  |  |
| --- | --- |
| **软件实现机制** | **示例** |
| **模块(Module)** | Routine, SubRoutine |
| **层(Layer)** | View, Logical, Model |
| **文件(File)** | DLL, EXE, DAT |
| **数据库(Database)** | Repository, Center Data |
| **进程(Process)** | Sender, Receiver |
| **网络节点(Physical Unit)** | Client, Server |

复合部件由更细粒度的部件和连接件组成，复合部件通过局部配置将其内部的部件和连接件连接起来，构成一个整体。



* **【连接件】**

连接件是软件体系结构中**承载部件之间交互的中介**，连接件并不直接关联到部件，这个工作将由配置来完成。

连接件包括抽象规格与具体实现两个部分。

抽象规格：包括角色和特征集。

特征集：包括类型、接口规则、交互断言、交互协议等。

角色：对外可见、可被外界引用的接口实体，每个角色代表一个交互参与方需要满足的一些条件，基本的条件是匹配该角色的端口所应复合的规则，复杂的条件可能会包括加密通信、负载均衡等。

* **【配置】**

部件和连接件都是软件体系结构独立的元素，相互之间没有直接的关联，**配置是将部件和连接件整合起来的专门的机制。**

配置通过部件端口与连接件角色相匹配的方式，将系统中部件和连接件的关系定义为一个关联集合，这个关联集合可以形成系统整体结构的一个拓扑描述。

利用配置将相互独立的部件和连接件联系起来，而不是之直接指定部件与连接件的关系的好处：

* 可以实现部件和连接件的一次定义，多处使用
* 在具体交互中，参与的部件不在固定，可以随时发生变化
* 对具体部件而言，其所参与的交互也不在固定，部件随时可以参与或退出一个交互
* **关注点——理解软件体系结构的三个方面之二**

**质量属性**是影响软件系统复杂度的关键因素，软件体系结构是处理质量属性和控制复杂度的主要手段，质量属性是软件体系结构最为重要的关注点。

软件体系结构的主要关注点：

**系统质量，项目环境，商业目标**

* **设计决策——理解软件体系结构的三个方面之三**

软件体系结构需要表现关注点，关注点对软件体系结构的最终的样式有着重要的影响，但是关注点并不能直接的反映为软件体系结构的高层结构元素，关注点与高层结构之间存在差距，关注点必须通过设计决策才能转化到高层结构上。

软件体系结构是重要设计决策的集合。

设计决策的定义：

设计决策对元素、特征和处理的选择，它们涉及一个或多个关注点，直接或间接的影响到软件体系结构。

设计决策的核心知识可以分为四个部分：关注点、解决方案、策略和理由。

**chapter 4+1 View**

* **逻辑视图**：关注系统的**逻辑结构**和**重要的设计机制**，描述系统提供的**功能和服务**。
* **开发视图**：关注系统的**实现结构**，描述系统**开发的组织**。
* **进程视图**：关注系统的**运行时表现**，描述系统的**并发进程组织**。
* **物理视图**：关注系统的**基础设施**，描述系统的**部署与分布**。
* **场景视图：**关注系统最为重要的**需求**，描述系统应该实现的**场景与用例**。
* **场景视图（用例视图）**

用例视图关注的是软件体系结构的需求，它们一方面说明软件体系结构设计的**出发点**，驱动其他4个视图的设计；另一方面用于**验证和评估**其他4个视图的设计，保证它们的正确性。

用例视图只是描述了软件体系结构的需求，并不涉及软件体系结构的抽象规格，也不涉及软件体系结构的实现，所以与其他4个视图不同的是，它没有直接贡献于软件体系结构自身的描述，以至于被称为“+1”。

用例视图中的用例既包括【功能需求用例】，还包括软件体系结构关注点【质量属性】和【项目环境】的用例。

* **逻辑视图**

用户角度考虑

逻辑视图描述的是一个满足了需求的软件体系结构设计，其主要内容是**软件体系结构的抽象规格**，主要关注点是满足用户的各项需求，尤其是**功能需求**、**质量属性需求和约束**。也就是说，逻辑视图从**总体功能组织**的角度来描述软件系统的高层结构，说明**用户需求在软件体系结构元素中的分解与分配**，解释软件系统保证需求实现的设计机制。

逻辑视图的内容是软件体系结构的抽象规格，其主要结构实体为部件、连接件、端口、角色、特征、配置等。

但是在描述复杂软件系统时，可能还需要详细描述部件与连接件的功能规格和交互协议。以对基本结构元素的扩展为基础，可以用下列机制来实现：

为部件类型建立行为状态图，描述部件类型的详细功能规格；

为连接件类型建立行为状态图，描述连接件类型的交互机制；

为端口建立协议状态机，描述部件端口和连接件角色的交互协议；

为配置和复合类型建立顺序图，描述配置和复合类型所包含的功能规格。

* **开发视图**

开发者角度考虑

开发视图关注软件体系结构的**模块实现**，体现软件系统的**模块组织**。

因此，软件体系结构的实现既要能够合理**划分**，建立模块化程度较好的模块集，以利于各个模块的独立开发与复用，又要能够将基础模块集组织成层、子系统等**层次结构**，以辅助项目管理活动的开展。

在逻辑视图的基础上，强调使用模块化和信息隐藏的思想**划分出易于开发实现**的模块。

每个模块对外存在需求或供给接口。

* **进程视图**

集成开发者角度考虑

进程视图关注软件体系结构的**运行时表现**，考虑在静态结构中难以分析的**性能、可靠性等非功能需求**，也会描述为保障系统完整性和容错性而需要的**进程并发及其分布**，还要说明软件体系结构的**抽象规格是如何被进程实现**的——可执行的进程、线程及其负责的操作与控制逻辑。

软件系统的功能可以划分为一系列相对独立的任务。在任务内部，实现元素之间的联系比较频繁，结合比较紧密。但是在任务之间，双方的实现元素之间只保持较少的连接，相对比较独立。因此，不同任务可能会被实现为不同的可执行单元——进程或者线程，这种分割实现的方式可以帮助降低软件系统的实现复杂度，所以在复杂的软件系统开发中是一种重要的设计机制。

进程就是由任务组形成的一个可执行单元，它是软件体系结构进程实现的基础元素。进程的并发与分布情况可以体现软件体系结构对性能和可靠性的处理机制。

* **部署视图**

系统工程师角度考虑

**将软件映射到硬件上**

部署视图关注可靠性、容错性、吞吐量、容量等**与基础设施相关的非功能需求**（例如，双机热备份系统的可靠性要高于单机系统，多机之间互相校验的系统要具有更高的容错性），描述软件体系结构的基础设施和网络分布，明确任务、进程、构件、重要制品（Artifact）以及环境的部署与分布。

**chapter 体系结构设计**

**体系结构设计的考虑方面**

结构

高层结构

多视图结构（4+1 View）

关注点

质量属性

项目环境

商业目标

设计决策

关注点产生的问题因素的解决方案

风格



一组设计决策形成的风格

**体系结构设计的步骤**

体系结构分析

体系结构需求分析

功能需求

关注点

质量属性

项目环境

商业目标

定义体系结构需求

定义需求的场景描述

定义初始体系结构

利用4+1 view表示

建立体系结构

确定设计决策，修正体系结构



**chapter 设计模式**

**面向接口编程**

通过模块的名字与整个模块建立的联系产生的耦合 < 与模块内部元素产生的耦合。

* **普通面向接口编程的手段**

封装

继承

* **集合类型中实现面向接口编程的手段**

1. **迭代器模式**

为遍历不同类型的集合提供统一的接口，隐藏了集合类型数据结构的内部结构信息。

体现了【面向接口编程】【信息隐藏】的思想。

集合类型的内部结构是可变的，不影响迭代器对外的接口。

迭代器对外的接口是可拓展的，可以随意添加集合类型的遍历方式。

1. **代理模式**

为其他对象提供了一种代理，来控制对目标对象的访问，通过代理实现间接的访问。

1. 原型模式？【集合类型中实现面向接口编程的手段？】

* **解耦的手段**

1. **避免重复**：重复的出现会导致一个地方的修改引发另一个地方也需要修改。
2. **DIP 依赖倒置原则**：高层模块不应该依赖于底层模块的实现，而是依赖于高层抽象。抽象不应该依赖与具体实现，具体实现应该依赖于抽象。
3. **间接访问**：代理模式
4. **协作设计**：中介者模式
5. **桥接模式**

* **中介者模式**

问题：

一组对象的交互遵从同一规则，但是十分复杂。

中介者模式将对象之间的交互封装起来，通过减少对象与对象之间的引用来解耦。

使得对象之间的交互可以独立变化。

【集中式控制风格】

* **桥接模式**

将接口与实现分离，提高了可拓展性，对客户端隐藏实现。

* **信息隐藏**

每个模块都拥有一个基本的秘密：外部行为和内部实现。

每个模块都隐藏了一个重要设计决策的实现，所以只有模块自身知道实现细节。

每个模块可能拥有一个额外的秘密：变化或修改。

* **一个模块信息隐藏的两种基本手段**
* **信息隐藏的两种基本类型：（隐藏设计决策、隐藏变化）**

1. **外观模式——隐藏设计决策**

外观模式提供了一个统一的接口，用来访问子系统中的一群接口。

它提供了一个高层接口，让子系统更容易的使用，隐藏了内部实现。

实现了客户端与子系统之间的解耦。

* 【外观模式】与【协作设计】的异同：

【外观模式】可以作为集中式的控制器。

【协作设计】封装的是由多个对象协作完成的一个task。

【外观模式】可以仅仅封装关系紧密的对象，不一定是封装形成一个task。

* 【外观模式】与【中介者模式】在协作设计方面的异同：

【外观模式】封装了关系紧密的几个对象，提供一个高层次的接口，供外界进行访问，主要作用是对外提供服务。

【中介者模式】将对象之间的交互进行封装，并不对外提供服务。

1. **策略模式——隐藏变化**

针对变化。

实现了可修改性和可复用性

* **共性和差异性**

继承关系：共性放在父类中，差异性放在子类中，实现多选一的关系。

聚合关系：共性放在整体中，差异性放在部分中，实现多选多的关系。

* **实现共性和可变性（差异性）的手段——隐藏变化？**

1. 策略模式
2. 状态模式
3. 桥接模式

* **OCP原则——隐藏变化？**

对拓展开放，对修改封闭。

对拓展开放：模块的行为是可以拓展的

对修改封闭：模块的内部实现的代码是不可修改的

1. **装饰者模式**

装饰者模式动态给一个对象添加一些额外的职责。

我们通常使用继承来实现功能的拓展，但如果需要拓展的功能的种类繁多，那么会生成很多子类，从而增加了系统的复杂性。同时，使用继承实现功能拓展，我们必须可预见这些拓展的功能，即功能在编译时就是确定的，是静态的。

使用装饰者模式，功能的拓展可以由用户动态决定加入的方式和时机，在运行期间决定何时增加何种功能，相比使用生成子类方式达到功能的扩充显得更为灵活。

1. **适配器模式**

* **运行时注册**

1. **观察者模式**
2. **命令模式**

* **对象创建**

1. **单体模式**
2. **工厂模式**
3. **抽象工厂模式**
4. **原型模式**

**chapter 职责分配（GRASP模式）**

**职责**：职责是对象的功能，维护对象对外的协议和状态。

* **职责对设计的影响：**

单一职责原则：每个模块只完成一个职责，实现一个功能。

信息隐藏：每个模块隐藏一个独立的设计决策。

module guide的来源：SRS和变化

职责分配：把职责分配给体系结构中的一个或多个类。

* **GRASP模式**
* **专家模式**

问题：面向对象设计中职责分配最基本的原则是什么？【职责应该分配给谁？】

解决：把职责分配给拥有完成该职责所必须的**信息**的类。

效果：

维护信息的封装

降低耦合

提高内聚

会导致一个类过于复杂

* **创建者模式**

问题：将创建对象的职责分配给谁？

解决：在决定哪个类负责对象的创建时，考虑潜在的创建者类和被创建的类之间的关系。

B负责创建A，如果：

B由A聚合而成；

B包含A；

（^B包含一个属性为A）

B中创建A供来使用；

B包含创建A的初始化信息。

效果：

通过让一个类负责创建它需要引用的对象，来降低耦合。

自己使用，自己创建，可以不依赖与其他类为自己创建。

* **低耦合模式**

问题：如何降低依赖，提高复用？

解决：通过职责分配降低耦合。

效果：

变化的部分可以被锁定。

容易理解，容易复用。

* **高内聚模式**

问题：如何维护复杂的管理？

解决：通过职责分配提高内聚。

效果：

类易于维护，易于理解，通常支持低耦合，支持复用。

* **控制者模式**

问题：处理系统外部事件的职责应该如何分配？

解决：如果系统从外部或图形化界面接收事件，那么添加一个响应事件的类来与处理事件的类解耦。

效果：

增加了复用性。使用控制器能够使得**外部的事件**与**内部的事件处理**相**独立**，不用考虑互相的类型或行为。

能够合理的解释用例的状态。确保了系统操作发生的**逻辑顺序**，使用例中活动或处理的状态得到合理的解释。

* **多态模式**

问题：如何处理类型的可变性？

解决：当行为是根据对象的类型变化时，使用多态的方法调用来选择行为，而不是利用if-else判断语句。

* **纯虚构模式**

问题：如何不违反高内聚、低耦合原则？

解决：把一组高内聚的职责分配给一个纯虚构的类，来支持高内聚、低耦合、易复用。

效果：

高内聚：关系紧密的相关任务构成的职责被分配到一个类中。

提高复用性。

* **间接模式**

问题：如何避免直接耦合？如何解耦？

解决：把职责分配给负责维护组件交互或服务的中介者，避免直接耦合。

例如：发布订阅风格或观察者模式。

* **Protected Variations保护可变性**

问题：如何设计对象、子系统或系统以保持变化性不会影响到其他元素？

解决：鉴别出可能发生变化或不稳定的元素，为他们创建一个稳定的接口。

* **职责的来源**

1. 用例中的系统行为

参与者与系统之间的交互

信息隐藏中的基本秘密

1. 软件体系结构设计

软件体系结构中的设计决策

信息隐藏中重要的次要秘密

1. 详细设计因素

详细设计决策

信息隐藏中的次要秘密

1. 特定的主题

对象的创建和撤销

关系的创建和撤销

数据的维护（初始化、保存和删除）

协作设计

* **职责分配注意事项**

确保对象的职责是紧密的：模块化。保持对象的行为与它拥有的信息一致。

确保对象没有做太多事情：单一职责原则。

职责不要重复：信息隐藏。

把一个复杂的职责进行分解。

**chapter 协作设计（控制结构）**

* **协作的含义：**

一个应用可以被划分为很多不同的行为。

每个行为由一组对象的协作实现。

每个协作都实现应用的一个行为。

在面向对象中，协作并不存在于一个独立的模块，而是分布在多个对象中。

* **协作设计**

1. **找出协作**
2. 用例中系统的行为
3. 软件体系结构设计中系统的行为

模块的接口

进程之间的通信

1. **设计协作**
2. **两个类之间的协作**

一个类向另外一个类发送消息。

使用委托，非控制：职责分配。

如果对象之间的协作是不可靠的，可以有一些保护方法：

1. 异常监控
2. 数据保护

手段：信息隐藏、只允许查询请求、使用安全代理、使用原型

1. 解耦

手段：依赖倒置、间接访问（委托、代理模式）、工厂模式

1. **系统行为之间的协作：控制结构**

分散式：系统行为的逻辑分布在整个对象形成的网络中。

集中式：系统行为的逻辑由一个独立的控制器控制。

集中式协作设计

控制器：做出决策并指导其他对象行为的对象。

与其他对象协作的原因：

聚集信息以做出决策；

调用其他对象的行为。

控制器只关注如何做出决策而不关注完成一个子系列的行为。

协调器：只用来传递信息和调用其他对象的方法。

协调器只关注与维护与对象之间的联系以用来传递信息和请求。

控制器把事情弄明白并采取行动。

协调器只做被告知要做的事情，基本不做决策。

* **协作（控制）风格**
* **分散式**：所有的系统行为逻辑分散在整个对象协作构成的网络中。

很多组件都拥有一些数据和一些职责。

控制流很难理解。

组件独立并不能完成很多工作，增加了耦合。

信息隐藏困难。

内聚性差。

几乎不满足模块化的原则。

* **集中式**：一些控制器集中控制者系统的行为逻辑。

容易找到决策是在哪里制定的；

容易知道决策是如何被制定的，并易于对决策制定的过程做出变化；

控制器可能非常复杂庞大，难以理解；

控制器可能把其他组件是为数据存储库：增加耦合，破坏信息隐藏。

* **委托式**：决策被分布在整个对象协作构成的网络中，有几个控制器做主要的决策。

控制器与少数的组件进行关联，降低耦合；

信息隐藏做的更好一些；

程序很容易进行分层；