一 软件危机表现、根源、解决方法

1. 软件危机的表现：

- 软件成本日益增长（相对数与绝对数）
- 开发进度难以控制（是一种高度智力的产品，不能采用人海战术）
- 软件质量差
- 软件维护困难

2. 软件危机产生的原因：

- 用户需求不明确
- 缺乏正确的理论指导
- 软件规模越来越大
- 软件复杂度越来越高

3. 软件危机的解决方案：

- 管理
- 采用工程化的开发方法
- 加大软件重用
- 采用先进的开发工具

二 软件体系结构的发展阶段及特征

“无体系结构”设计阶段 ⇦ 以汇编语言进行小规模应用程序开发为特征

萌芽阶段 ⇦ 出现了程序结构设计主题，以控制流图和数据流图构成软件结构为特征

初期阶段 ⇦ 出现了从不同侧面描述系统的结构模型，以UML为典型代表。

高级阶段 ⇦ 以描述系统的高层抽象结构为中心，不关心具体的建模细节，划分了体系结构模型与传统软件结构的界限，该阶段以Kruchten提出的“4+1”模型为标志

三 软件体系结构三种定义及其含义

1. Dewayne Perry 和 A1exander Wo1f

软件体系结构是具有一定形式的结构化元素，即**构件的集合**，包括处理构件、数据构件和连接构件。

处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组合连接起来。

2. Mary Shaw 和 David Garlan

软件体系结构是软件设计过程中的一个层次，这一层次超越计算过程中的算法设计和数据结构设计。体系结构问题**包括总体组织和全局控制、通讯协议、同步、数据存取，给设计元素分配特定功能，设计元素的组织，规模和性能，在各设计方案间进行选择等**。

软件体系结构处理算法与数据结构之上关于整体系统结构设计和描述方面的一些问题，如全局组织和全局控制结构、关于通讯、同步与数据存取的协议，设计构件功能定义，物理分布与合成，设计方案的选择、评估与实现等。

3. Kruchten

软件体系结构有四个角度，它们从不同方面对系统进行描述：概念角度描述系统的主要构件及它们之间的关系；模块角度包含功能分解与层次结构；运行角度描述了一个系统的动态结构；代码角度描述了各种代码和库函数在开发环境中的组织。

4. Hayes Roth

软件体系结构是一个抽象的系统规范，主要**包括用其行为来描述的功能构件和构件之间的相互连接、接口和关系**。

5. David Garlan 和 Dewne Perry

软件体系结构是一个程序／系统各构件的结构、它们之间的相互关系以及进行设计的原则和随时间演化的指导方针。

6. Barry Boehm

软件体系结构包括一个软件和系统构件，互联及约束的集合；一个系统需求说明的集合；一个基本原理用以说明这一构件，互联和约束能够满足系统需求。

7. Bass，Ctements 和 Kazman

软件体系结构包括一个或一组软件构件、软件构件的外部的可见特性及其相互关系。其中，“软件外部的可见特性”是指软件构件提供的服务、性能、特性、错误处理、共享资源使用等。

8. 张友生：

软件体系结构为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素的描述、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。软件体系结构不仅指定了系统的组织结构和拓扑结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。

特点：从高层抽象了软件系统结构、行为、属性。

四 软件体系结构意义

- 体系结构是风险承担者（又称为涉众，stakeholder）进行交流的手段
- 体系结构是早期设计决策的体现
- 软件体系结构是可重用的模型

五 软件体系结构几种模型的基本含义

1. 结构模型

这是一个最直观、最普遍的建模方法。这种方法以体系结构的构件、连接件和其他概念来刻画结构，并力图通过结构来反映系统的重要语义内容，包括系统的配置、约束、隐含的假设条件、风格、性质等。

研究结构模型的核心是体系结构描述语言。

2. 框架模型

框架模型与结构模型类似，**但它不太侧重描述结构的细节而更侧重于整体的结构**。

框架模型主要以一些特殊的问题为目标建立只针对和适应该问题的结构。

3. 功能模型

功能模型认为体系结构是由一组功能构件**按层次组成**，下层向上层提供服务。

功能模型可以看作是一种特殊的框架模型。

4. 动态模型

动态模型是对结构或框架模型的**补充**，研究系统的“**大颗粒**”的行为性质。

例如，描述系统的重新配置或演化。动态可以指系统总体结构的配置、建立或拆除通信通道或计算的过程。

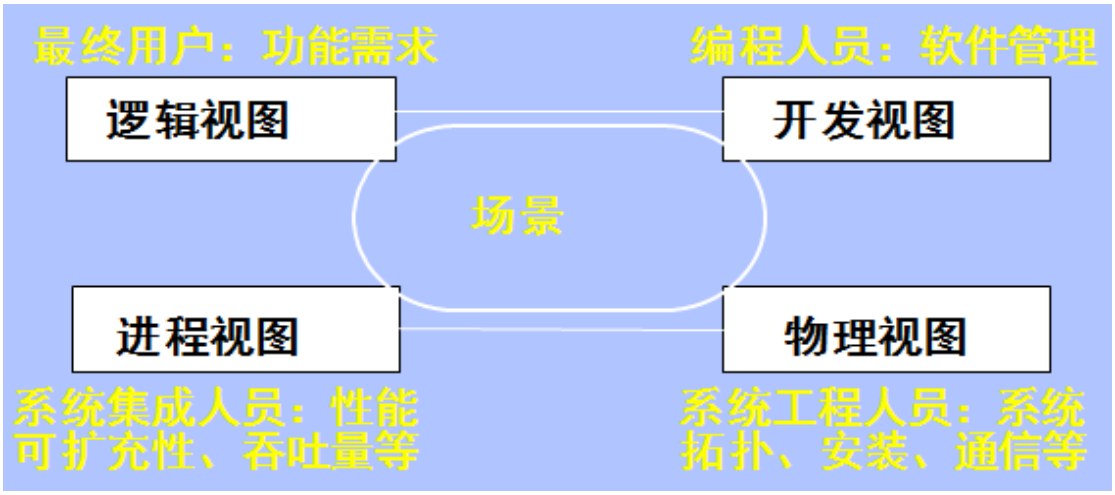
5. 过程模型

过程模型研究构造系统的步骤和过程。

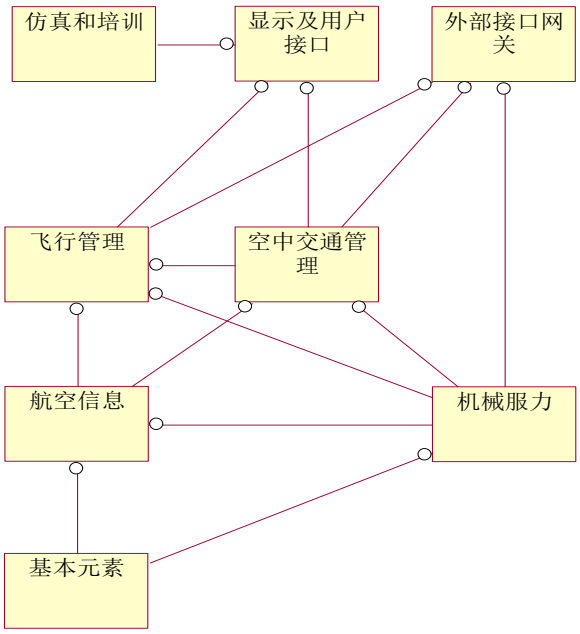
结构是遵循某些过程脚本的结果。

六 (!重点)软件体系结构 4+1 视图模型含义及设计(重点!)

0. “4+1”模型概述



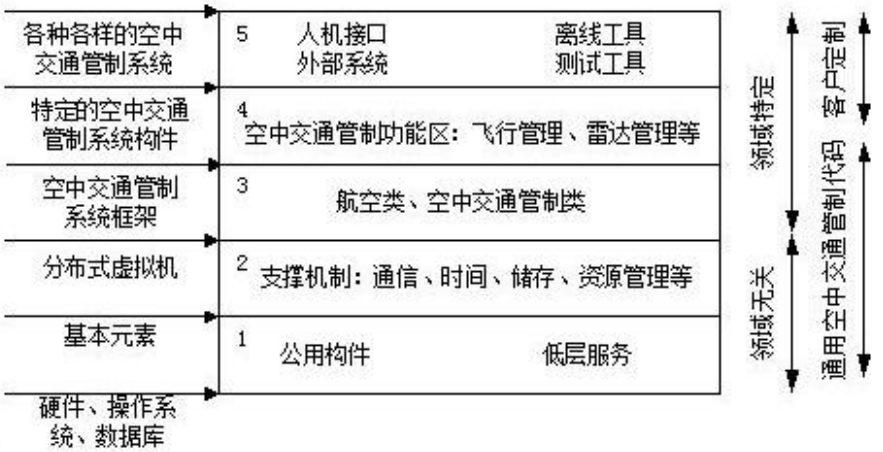
- 示意图：
- 含义：“4+1”视图模型从 5 个不同的视角包括逻辑视图、进程视图、物理视图、开发视图和场景视图来描述软件体系结构。每一个视图只关心系统的一个侧面，5 个视图结合在一起才能反映系统的软件体系结构的全部内容。



1. 逻辑视图

- 逻辑视图主要支持系统的功能需求，即系统提供给最终用户的服务。在逻辑视图中，系统分解成一系列的功能抽象，这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析，而且可用作标识在整个系统的各个不同部分的通用机制和设计元素。
- 在面向对象技术中，通过抽象、封装和继承，可以用**对象模型**来代表逻辑视图，用**类图**来描述逻辑视图。
- 设计：逻辑视图中使用的***风格***为**面向对象的风格**，逻辑视图设计中要注意的主要问题是要保持一个单一的、内聚的对象模型贯穿整个系统。

2. 开发视图



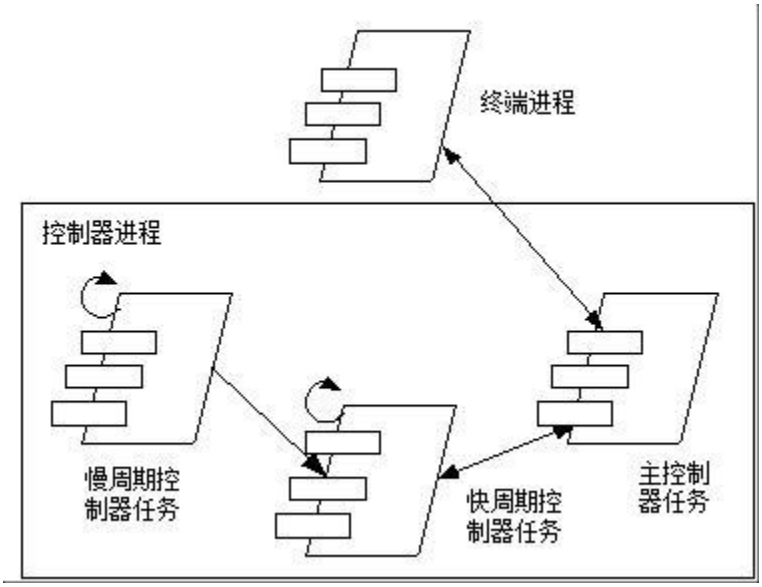
- 开发视图也称模块视图，主要***侧重***于软件模块的**组织和管理**。
- 开发视图要考虑**软件内部**的需求，如软件开发的容易性、软件的重用和软件的通用性，要充分考虑由于具体**开发工具的不同**而带来的**局限性**。
- 开发视图**通过系统输入输出关系**的模型图和子系统图来描述。
- 设计：在开发视图中，最好采用 **4-6 层**子系统，而且每个子系统仅仅能与**同层或**

更低层的子系统通讯，这样可以使每个层次的接口既完备又精练，**避免**了各个模块之间很**复杂**的**依赖关系**。

- 设计时要充分考虑，对于各个层次，**层次越低**，**通用性越强**，这样，可以保证应用程序的需求发生改变时，所做的改动最小。开发视图所用的***风格***通常是**层次结构风格**。

3. 进程视图

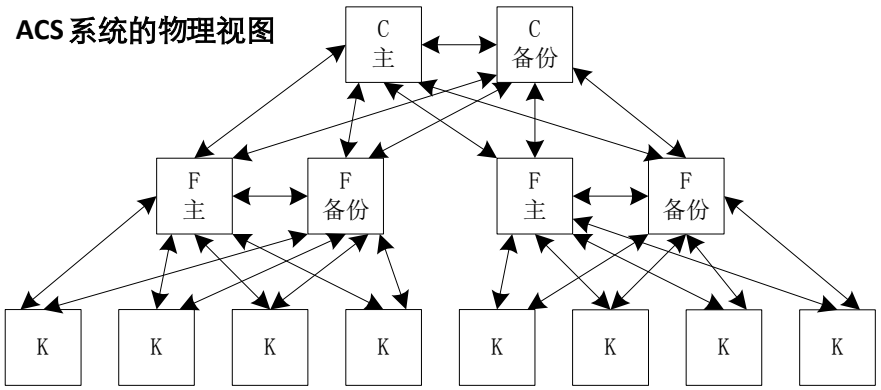
- 进程视图***侧重***于系统的**运行特性**，主要关注一些**非功能性**的需求。
- 进程视图强调**并发性**、**分布性**、**系统集成性**和**容错能力**，以及从逻辑视图中的主要抽象如何适合进程结构。它也定义逻辑视图中的各个类的操作具体是在哪一个线程中被执行的。
- 设计：进程视图可以描述成**多层抽象**，每个级别分别**关注不同的方面**。在最高层抽象中，进程结构可以看作是构成一个执行单元的一组任务。它可看成一系列独立的，通过逻辑网络相互通信的程序。它们是分布的，通过总线或局域网、广域网等硬件资源连接起来。



4. 物理视图

- 物理视图主要考虑如何把软件映射到硬件上，它通常要考虑到系统性能、规模、可靠性等。解决系统拓扑结构、系统安装、通讯等问题。
- 当软件运行于不同的节点上时，各视图中的构件都直接或间接地对应于系统的不同节点上。因此，从软件到节点的映射要有较高的灵活性，当环境改变时，对系统其他视图的影响最小。

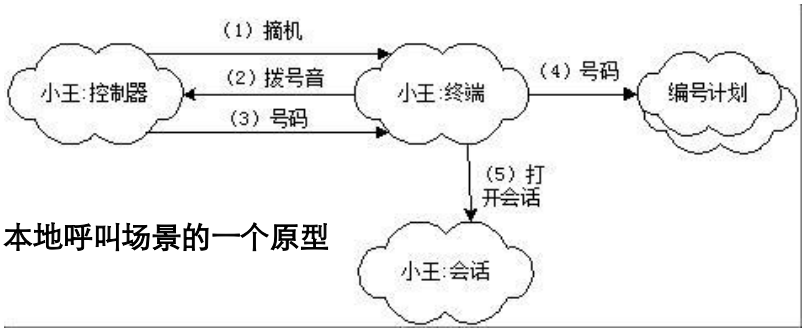
ACS 系统的物理视图



- 设计：大型系统的物理视图可能会变得十分混乱，因此可以与进程视图的映射一道，以多种形式出现，也可单独出现。

5. 场景

- 场景可以看作是那些重要系统活动的抽象，它使四个视图有机联系起来，从某种意义上说场景是最重要的需求抽象。在开发体系结构时，它可以帮助设计者找到体系结构的构件和它们之间的作用关系。同时，也可以用场景来分析一个特定的视图，或描述不同视图构件间是如何相互作用的。



本地呼叫场景的一个原型

- 场景可以用文本表示，也可以用图形表示。

6. 设计思想：

- 逻辑视图和开发视图描述系统的静态结构，而进程视图和物理视图描述系统的动态结构。
- 对于不同的软件系统来说，侧重的角度也有所不同。例如，对于管理信息系统来说，比较侧重于从逻辑视图和开发视图来描述系统，而对于实时控制系统来说，则比较注重于从进程视图和物理视图来描述系统。

七 软件体系结构风格定义、含义

1. 定义与含义：

- 软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式。
- 体系结构风格定义了一个系统家族，即定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型，而这组约束指出系统是如何将这些构件和连接件组合起来的。
- 体系结构风格反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。

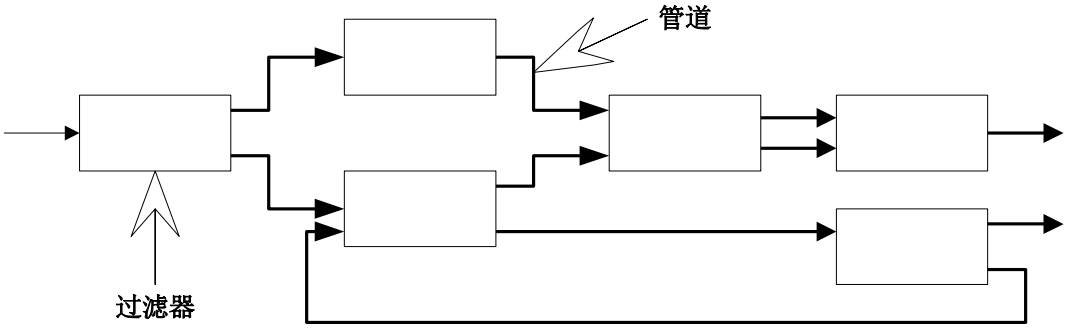
2. 经典的体系结构风格 (Garlan, Shaw)

- 数据流风格：批处理序列；管道/过滤器。
- 调用/返回风格：主程序/子程序；面向对象风格；层次结构。
- 独立构件风格：进程通讯；事件系统。
- 虚拟机风格：解释器；基于规则的系统。
- 仓库风格：数据库系统；超文本系统；黑板系统。

八 几种经典的体系结构风格（管道过滤器、面向对象风格、隐式调用风格、仓库风格、CS、BS、层次风格、异构风格）的组成、结构、优缺点、例子

1. 管道/过滤器

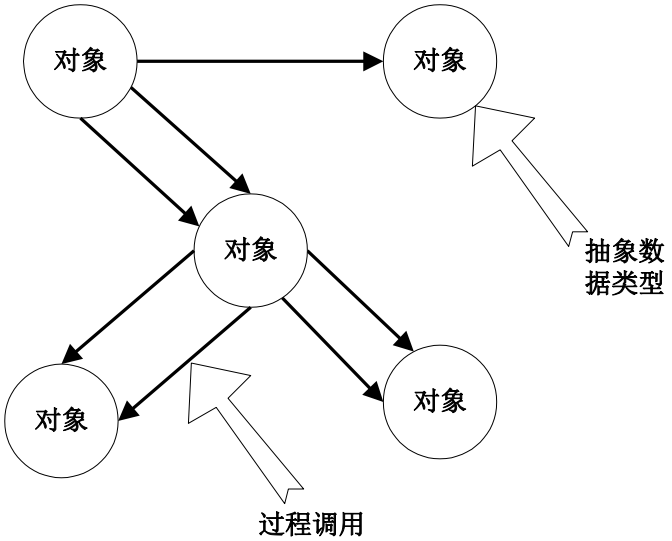
- 含义：每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全消费之前，输出便产生了。
  这里的构件被称为过滤器，这种风格的连接件就象是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。
- 组成：**构件（过滤器）和连接件（管道）**



- 结构：
- 优点：
  - 使得软构件具有良好的**隐蔽性**和**高内聚**、**低耦合**的特点；
  - 允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的**简单合成**；
  - 支持**软件重用**。只要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来；
  - 系统**维护**和增强系统性能**简单**。新的过滤器可以添加到现有系统中来；旧的可以被改进的过滤器替换掉；
  - 允许对一些如吞吐量、死锁等属性的分析；
  - 支持**并行**执行。每个过滤器是作为一个单独的任务完成，因此可与其它任务并行执行。
- 缺点：
  - 通常导致进程成为批处理的结构。
  - **不适合**处理**交互**的应用。
  - 因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这样就导致了系统**性能下降**，并增加了编写过滤器的复杂性。
- 例子：Unix，DOS 中的重定向： dir | sort（perl 脚本语言）编译器

2. 数据抽象和面向对象组织

- 含义：这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作**封装**在一个抽象数据类型或对象中。这种风格的构件是对象，或者说是抽象数据类型的实例。对象是一种被称作管理者的构件，因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。
- 组成：**构件（对象）**



- 结构：
- 优点：
  - 因为对象对其它对象隐藏它的表示，所以可以**改变**一个对象的表示，而**不影响其它的对象**；
  - 设计者可将一些**数据存取操作**的问题**分解成**一些**交互**的**代理程序**的**集合**。

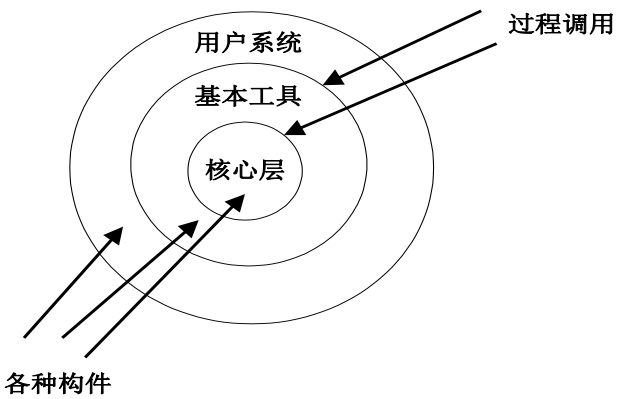
- 缺点：
  - 为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的**标识改变**了，就**必须修改**所有**其他明确调用它的对象**；
  - **必须修改所有显式调用它的其它对象**，并消除由此带来的一些副作用。例如，如果 A 使用了对象 B，C 也使用了对象 B，那么，C 对 B 的使用所造成的对 A 的影响可能是料想不到的。

3. 基于事件的隐式调用

- 含义：构件不直接调用一个过程，而是触发或广播一个或多个事件。系统中的其它构件中的过程在一个或多个事件中注册，当一个事件被触发，系统自动调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。这种风格的构件是一些模块，模块既可以是一些过程，又可以是一些事件的集合。过程可以用通用的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被调用。这种风格的主要特点是事件的触发者并不知道哪些构件会被这些事件影响。这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用，因此，许多隐式调用的系统也包含显式调用作为构件交互的补充形式。
- 组成：构件（一些**模块**，模块既可以是一些过程，又可以是一些事件的集合）
- 优点：
  - 为软件**重用**提供了**强大**的**支持**。当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中。
  - 为**改进系统**带来了**方便**。当用一个构件代替另一个构件时，不会影响到其它构件的接口。
- 缺点：
  - 构件**放弃**了**对系统**计算的**控制**。一个构件触发一个事件时，不能确定其它构件是否会响应它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。
  - 数据交换**效率**的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这些情况下，全局性能和资源管理便成了问题。
  - 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。

4. 分层系统

- 含义：层次系统组织成一个层次结构，每一层为上层服务，并作为下层客户。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中构件在一些层实现了虚拟机（在另一些层次系统中层是部分不透明的）。连接件通过决定层间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。这种风格支持基于可增加抽象层的设计。允许将一个复杂问题分解成一个增量步骤序列的实现。由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件重用提供了强大的支持。
- 组成：构件，连接件

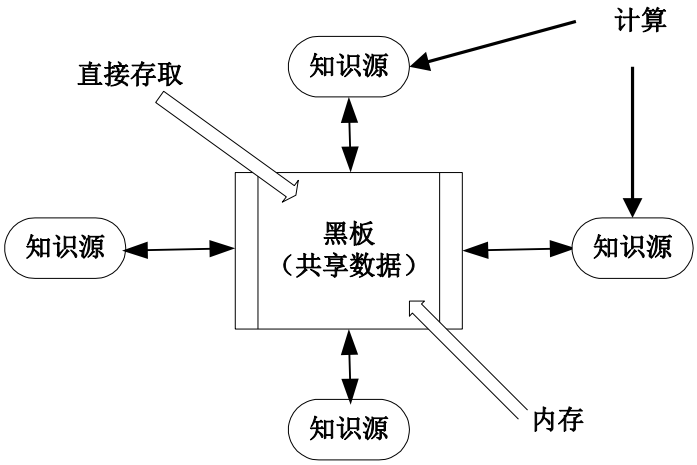


- 结构：

- 优点：
  - 支持基于**抽象程度递增**的系统**设计**，使设计者可以把一个复杂系统按递增的步骤进行分解；
  - 支持**功能增强**，因为每一层至多和相邻的上下层交互，因此功能的改变最多影响相邻的上下层；
  - 支持**重用**。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。
- 缺点：
  - 并**不**是每个系统都可以很**容易**地**划分**为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计师不得不把一些低级或高级的功能综合起来；
  - **很难找到**一个合适的、正确的**层次抽象方法**。

5. 仓库系统及知识库

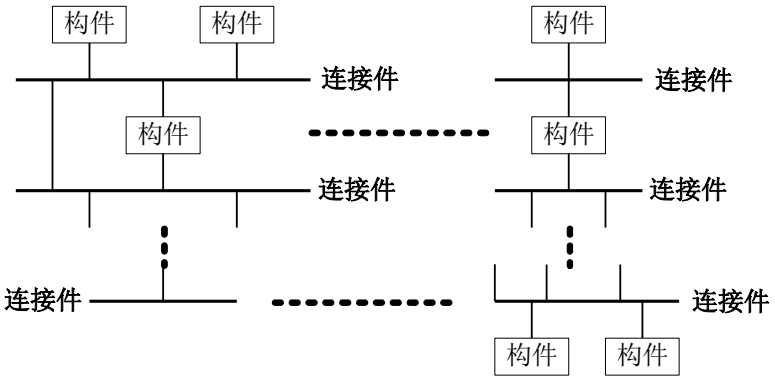
- 含义：在仓库风格中，有两种不同的构件：中央数据结构说明当前状态，独立构件在中央数据存贮上执行，仓库与外构件间的相互作用在系统中会有大的变化。控制原则的选取产生两个主要的子类。若输入流中某类时间触发进程执行的选择，则仓库是一传统型数据库；另一方面，若中央数据结构的当前状态触发进程执行的选择，则仓库是一黑板系统。
- 组成：构件（中央数据结构，独立构件）



- 结构：
- 例子：黑板系统的传统应用是信号处理领域，如语音和模式识别。另一个应用是松耦合代理数据共享存取。

6. C2 风格

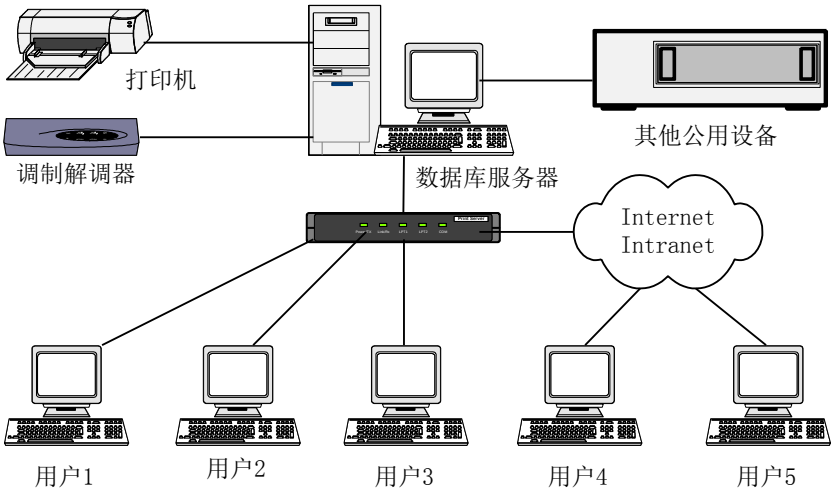
- 含义：通过连接件绑定在一起的按照一组规则运作的并行构件网络。C2 风格中的系统组织规则如下：
  - 系统中的构件和连接件都有一个顶部和一个底部；
  - 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，而构件与构件之间的直接连接是不允许的；
  - 一个连接件可以和任意数目的其它构件和连接件连接；
  - 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。
- 组成：构件、连接件



- 结构：
- 特点：
  - 系统中的构件可**实现应用需求**，并能将**任意复杂度的功能封装在一起**；
  - 所有构件之间的通讯是通过以连接件为中介的**异步消息交换机制**来实现的；
  - **构件相对独立**，构件之间**依赖性较少**。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

7. C/S

- 含义：C/S 软件体系结构是基于资源不对等，且为实现共享而提出来的，是 20 世纪 90 年代成熟起来的技术，C/S 体系结构定义了工作站如何与服务器相连，以实现数据和应用分布到多个处理机上。
  C/S 体系结构有三个主要组成部分：数据库服务器、客户应用程序和网络。
- 组成：**数据库服务器、客户应用程序和网络**



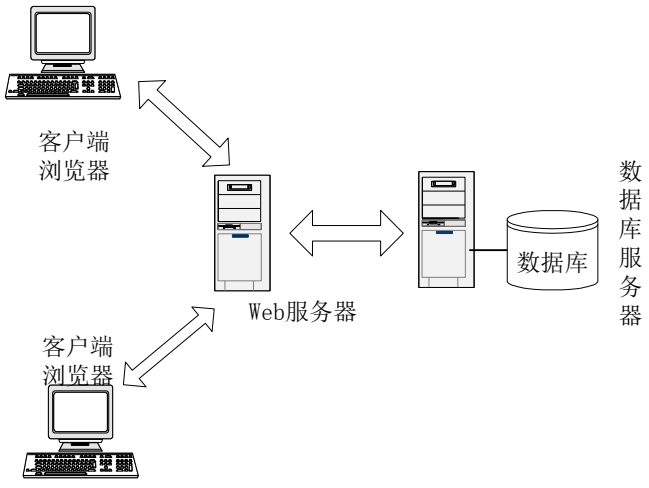
- 结构：
- 优点：
  - 模型**思想简单**，**易**于人们**理解**和**接受**。
  - **灵活、易维护与扩充**：系统的客户应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以适合各构件的要求，这对于硬件和软件的变化显示出极大的适应性和灵活性，而且易于对系统进行扩充和缩小。
  - **资源**可以进行**合理配置**：在 C/S 体系结构中，系统中的**功能构件充分隔离**，客户应用程序的开发集中于数据的显示和分析，而数据库服务器的开发则集中于数据的管理，不必在每一个新的应用程序中都要对一个 DBMS 进行编码。将大的应用处理任务分布到许多通过网络连接的低成本计算机上，以节约大量费用。
- 缺点：
  - 开发**成本**较**高**
  - 客户端程序**设计复杂**

- 用户界面**风格不一**，**使用繁杂**，不利于推广使用
- 软件**移植困难**
- 软件**维护**和**升级困难**
- 新技术不能轻易应用

- 例子：QQ、MSN、飞信、锐捷客户端、HomeShare 网络共享软件。

8. B/S

- 含义：浏览器/服务器（B/S）风格就是上述三层应用结构的一种实现方式，其具体结构为：浏览器/Web 服务器/数据库服务器。B/S 体系结构主要是利用不断成熟的 WWW 浏览器技术，结合浏览器的多种脚本语言，用通用浏览器就实现了原来需要复杂的专用软件才能实现的强大功能，并节约了开发成本。从某种程度上来说，B/S 结构是一种全新的软件体系结构。
- 组成：浏览器/Web 服务器/数据库服务器



- 结构：
- 优点：
  - 基于 B/S 体系结构的软件，系统安装、修改和维护全在服务器端解决。用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，真正达到了**“零客户端”**的**功能**，很**容易**在运行时自动**升级**。
  - B/S 体系结构还提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的**开放性基础**。
- 缺点：
  - 没有集成有效的数据库处理功能，对**数据处理功能不强**。
  - **安全**性**难**以**控制**。
  - 采用 B/S 体系结构的应用系统，在数据查询等**响应速度**上，要远远地**低**于 C/S 体系结构。
  - B/S 体系结构的数据提交一般以页面为单位，数据的**动态交互**性**不强**，不利于在线事务处理(OLTP)应用

9. 为什么要使用异构风格

- **不同的结构有不同的处理能力的强项和弱点**，一个系统的体系结构应该根据实际需要进行选择，以解决实际问题。
- **关于软件包、框架、通信以及其他一些体系结构上的问题，目前存在多种标准**。即使在某段时间内某一种标准占统治地位，但变动最终是绝对的。
- **一些遗留下来的代码**，它们仍有效用，但是却与新系统有某种程度上的不协调。然而在许多场合，将技术与经济综合进行考虑时，总是决定不再重写它们。

九 体系结构描述的几种方法（图形表达工具、模块互连接语言(MIL)、基于软构件的系统描述语言、基于 UML 对体系结构进行建模、软件体系结构描述语言(ADL)）的基本内容以及各自优缺点

1. 图形表达工具

2. 模块互连接语言

- 基本内容：采用将**一种或几种**传统程序设计**语言**的**模块**连接起来的模块互连接语言。
- 优点：
  - 由于程序设计语言和模块内连接语言具有严格的**语义基础**，因此它们能支持对较大的软件单元进行描述，诸如定义/使用和扇入/扇出等操作。例如，Ada 语言采用 use 实现包的重用，Pascal 语言采用过程（函数）模块的交互等。
  - MIL 方式**对**模块化的程序设计和分段编译等**程序设计**与**开发技术**确实**发挥**了**很大**的**作用**。
- 缺点：但是由于这些语言处理和描述的软件设计开发层次**过于依赖**程序**设计语言**，因此限制了它们处理和描述比程序设计语言元素更为抽象的高层次软件体系结构元素的能力。

3. 基于软构件的系统描述语言

- 基本内容：基于软构件的系统描述语言将软件系统描述成一种是由许多以特定形式相互作用的特殊软件实体构造组成的组织或系统。
- 优点：
  - 可以用来在一个较高的抽象层次上对系统的体系结构建模，Darwin 最初用作设计和构造复杂分布式系统的配置说明语言，因具有动态特性，也可用来描述动态体系结构。
  - 一种**以构件为单位**的软件系统描述方法。
- 缺点：但是他们所面向和针对的系统元素仍然是一些层次较低的以程序设计为基础的通信协作软件实体单元，而且这些语言所描述和表达的系统一般而言都是面向特定应用的特殊系统，这些特性使得基于软构件的系统描述仍然不是十分适合软件体系结构的描述和表达。

4. 基于 UML 对体系结构进行建模

- 缺点：
- UML **缺乏**对体系结构下述的元素进和相应的描述与应用能力
  - 体系结构风格
  - 显式的体系结构连接器
  - 体系结构约束
- 总体来讲：UML 是一种非形式化的描述语言，缺乏严格的语意描述，不能表达体系结构中的语义，不能描述体系结构的相关模型。
- UML2 有所改进。

5. 软件体系结构描述语言(ADL)

- 基本内容：ADL 是在底层语义模型的支持下，为软件系统的概念体系结构建模提供了具体语法和概念框架。基于底层语义的工具为体系结构的表示、分析、演化、细化、设计过程等提供支持。其三个基本元素是：构件、连接件、体系结构配置。
- 优点：　（跟其他语言的比较）
  - 构造能力：ADL 能够使用较小的独立体系结构元素来建造大型软件系统；
  - 抽象能力：ADL 使得软件体系结构中的构件和连接件描述可以只关注它们的抽象特性，而不管其具体的实现细节；
  - 重用能力：ADL 使得组成软件系统的构件、连接件甚至是软件体系结构都成为软件系统开发和设计的可重用部件；
  - 组合能力：ADL 使得其描述的每一系统元素都有其自己的局部结构，这种描述局部结构的特点使得 ADL 支持软件系统的动态变化组合；
  - 异构能力：ADL 允许多个不同的体系结构描述关联存在；
  - 分析和推理能力：ADL 允许对其描述的体系结构进行多种不同的性能和功能上的多种推理分析。
- 缺点：
  - 现有的 ADL 大多是**与领域相关**的，这**不利于**对**不同领域**体系结构的**说明**。
  - 这些针对于不同领域的 ADL 在某些方面又**大同小异**，造成了**资源**的**冗余**。
  - 有些 ADL 可以实现构件与连接件的演化，但这样的**演化能力**是**有限**的，这样的演化大多是通过子类型实现的。而且，系统级的演化能力才是最终目的。
  - 尽管现有的 ADL 都提供了支持工具集，但将这些 ADL 与工具**应用于实际**系统开发中的**成功**范例还**有限**。
  - 支持工具的**可用性**与**有效性较差**，严重地阻碍了这些 ADL 的广泛应用。

十 (!重点)利用 C2、ACME 语言描述方法描述系统(重点!)

1. C2

- 概述：
  - C2 和其提供的设计环境（Argo）支持采用基于时间的风格来描述用户界面系统，并支持使用可替换、可重用的构件开发 GUI 的体系结构。
  - 在 C2 中，连接件负责构件之间消息的传递，而构件维持状态、执行操作并通过两个名字分别为“*top*”和“*bottom*”的端口和其它的构件交换信息。
  - 每个接口包含一种可发送的消息和一组可接收的消息。构件之间的消息要么是请求其它构件执行某个操作的请求消息，要么是通知其他构件自身执行了某个操作或状态发生改变的通知消息。
  - 构件之间的消息交换不能直接进行，而只能通过连接件来完成。每个构件接口最多只能和一个连接件相连，而连接件可以和任意数目的构件或连接件相连。
  - 请求消息只能向上层传送而通知消息只能向下层传送。
  - 通知消息的传递只对应于构件内部的操作，而和接收消息的构件的需求无关。
  - C2 对构件和连接件的实现语言、实现构件的线程控制、构件的部署以及连接件使用的通讯协议等都不加限制。

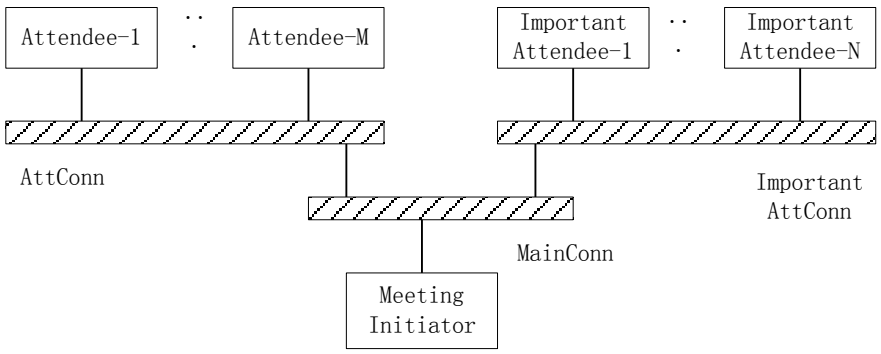
- C2 对构件接口的描述：

```
Component ::=
    component component_name is
        interface component_message_interface
        parameters component_parameters
        methods component_methods
        [behavior component_behavior]
        [context component_context]
     end component_name;
```

- C2 对构件的描述

```
component_message_interface ::=
   top_domain_interface
   bottom_domain_interface

top_domain_interface ::=
   top_domain is
      out interface_requests
      in interface_notifications

bottom_domain_interface ::=
   bottom_domain is
      out interface_notifications
      in interface_requests
interface_requests ::=
   {request; } | null;

interface_notifications ::=
   {notification; } | null;

request ::=
   message_name(request_parameters)

request_parameters ::=
   [to component_name][parameter_list]

notification ::=
   message_name[parameter_list]
```

- 会议安排系统的 C2 风格



- C2 对会议安排系统的描述

```
system MeetingScheduler_1 is
  architecture MeetingScheduler with
    Attendee instance Att_1, Att_2, Att_3;
    ImportantAttendee instance ImpAtt_1, ImpAtt_2;
    MeetingInitiator instance MtgInit_1;
end MeetingScheduler_1;
```

2. ACME

- 概述
  - 是一种简单的、通用的体系结构描述语言，提从了一种在不同的体系结构的实例之间的变换机制。
  - 四个不同方面进行描述软件体系结构：结构、属性、约束、类型与风格。
  - 七个描述类型的核心实体：系统，组件，连接器，端口，角色，展现(representations)，展现映射 rep-maps (representation map 的简写)
- ACME 的属性，约束
  - 属性：
    - ACME 利用属性对体系结构进行说明，可用来记录设计的细节
    - 每个属性有名字，类型和值
  - 约束：
    - 一种特殊的属性，
    - 基于一阶谓词
    - 有 invariant（不变式）与 heruistic（启发式）两种形式
- ACME 程序：

```
System simple_cs = {
   Component client = {
     Port send-request;
     Property Aesop-style : style-id = client-server;
     Property UniCon-style : style-id = client-server;
     Property source-code : external = "CODE-LIB/client.c";
   }
   Component server = {
     Port receive-request;
     Property idempotence : boolean = true;
     Property max-concurrent-clients : integer = 1;
```

```
    source-code : external = "CODE-LIB/server.c";
  }
  Connector rpc = {
    Role caller;
    Role callee;
    Property asynchronous : boolean = true;
    max-roles : integer = 2;
    protocol : Wright = " ... ";
  }
  Attachment client.send-request to rpc.caller;
  Attachment server.receive-request to rpc.callee;
}
```

- ACME 类型与风格
  - 属性的类型
  - 结构类型：结构元素的类型
  - 系统风格：在 ACME 中称为 family，通过所含有的元素，它们之间的约束，它们之间的默认结构来定义。而对于满足某种风格的具体的系统，它应满足风格的一些约定。

十一 设计模式定义、作用、利用设计模式设计方法

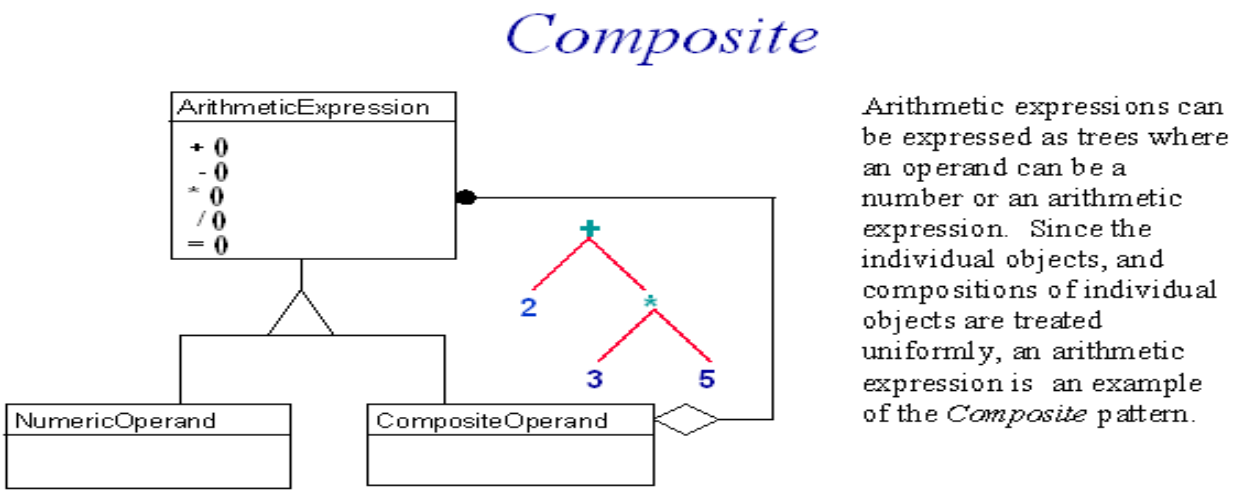
- 软件设计模式定义：
  - 对通用设计问题的重复解决方案
  - 对真实世界问题的实践的/具体的解决方案
  - 面向特定的问题环境
  - 权衡利弊之后得到的“最佳”解决方案
  - 领域专家和设计老手的“杀手锏”
  - 用文档的方式记录的最佳实践
  - 在讨论问题的解决方案时，一种可交流的词汇
  - 在使用（重用）、共享、构造软件系统中，一种有效地使用已有的智慧/经验/专家技术的方式
- 设计模式的作用：利用设计模式可方便地重用成功的设计和结构。把已经证实的技术表示为设计模式，使他们更加容易被新系统的开发者所接受。设计模式帮助设计师选择可使系统重用的设计方案，避免选择危害到可重用性的方案。设计模式还提供了类和对象接口的明确的说明书和这些接口的潜在意义，来改进现有系统的记录和维护。

十二 (!重点)几种常见模式(如组合模式、适配器模式、工厂方法模式)的基本使用场景与设计方法(重点!)

1. 组合模式(Composite Pattern)：

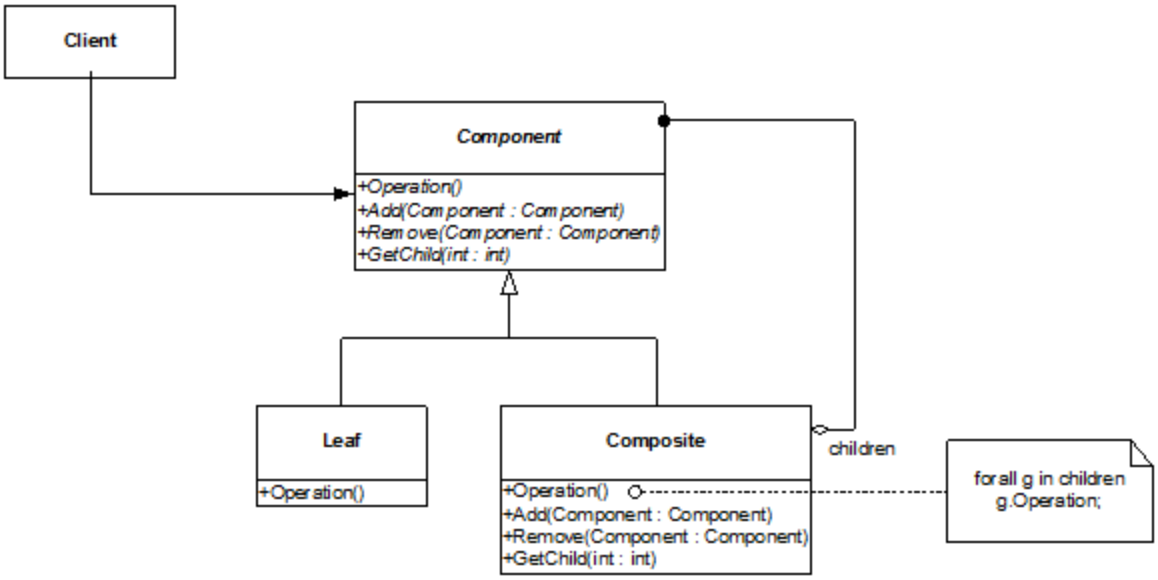
- **意图：**将对象组合成树形结构以表示“**部分-整体**”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

- **例如**，算术表达式能够用树结构来表示，其中操作数可以是一个数字或者一个算术表达式。如果我们把单个对象(数字)，和单个对象的组合(子数学表达式)同等地对待，这个算术表达式就是一个 *Composite* 模式的例子。

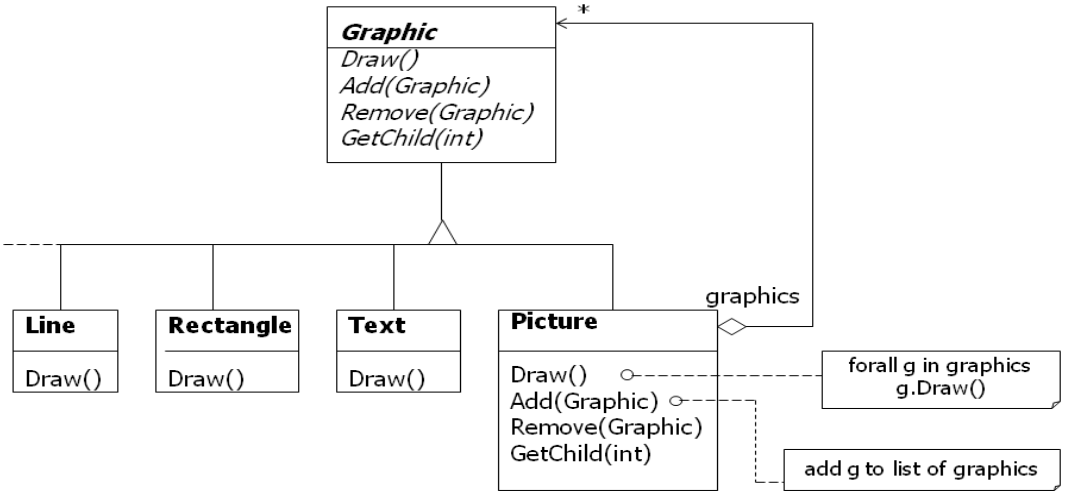


Structural

- 结构:

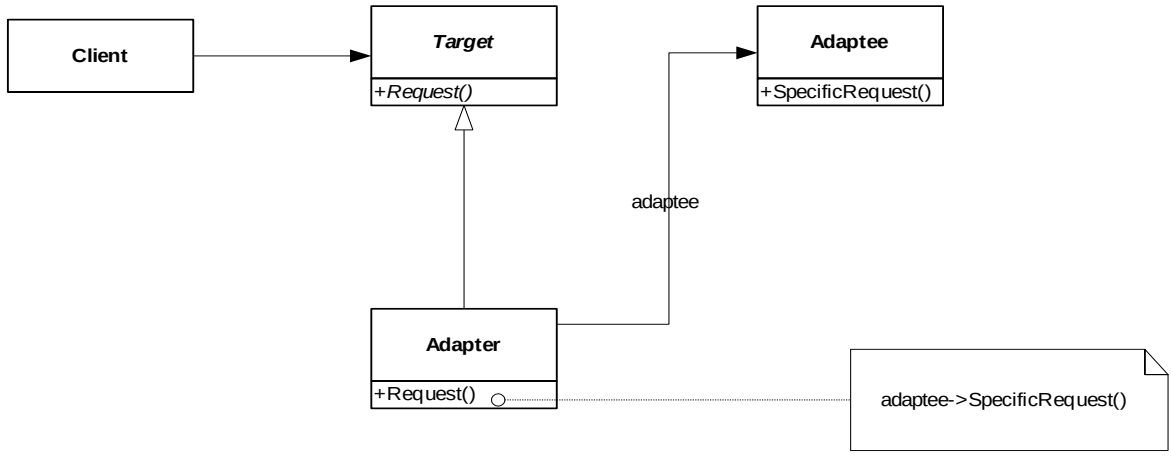


- 应用例子:



2. Adapter（适配器）

- 意图：将一个类的接口**转换成**用户希望的另一个接口。使得**原来**由于接口**不兼容**而不能在一起工作的那些类**可以**在**一起工作**。
- 例如，一个 1/2 英寸的防倒转齿轮不可能和一个 1/4 英寸的插槽配合工作。使用 *Adapter* 模式，阴口和 1/2 英寸的防倒转齿轮接合，阳口和 1/4 英寸的插槽接合。这样两个部件就能够顺利工作。

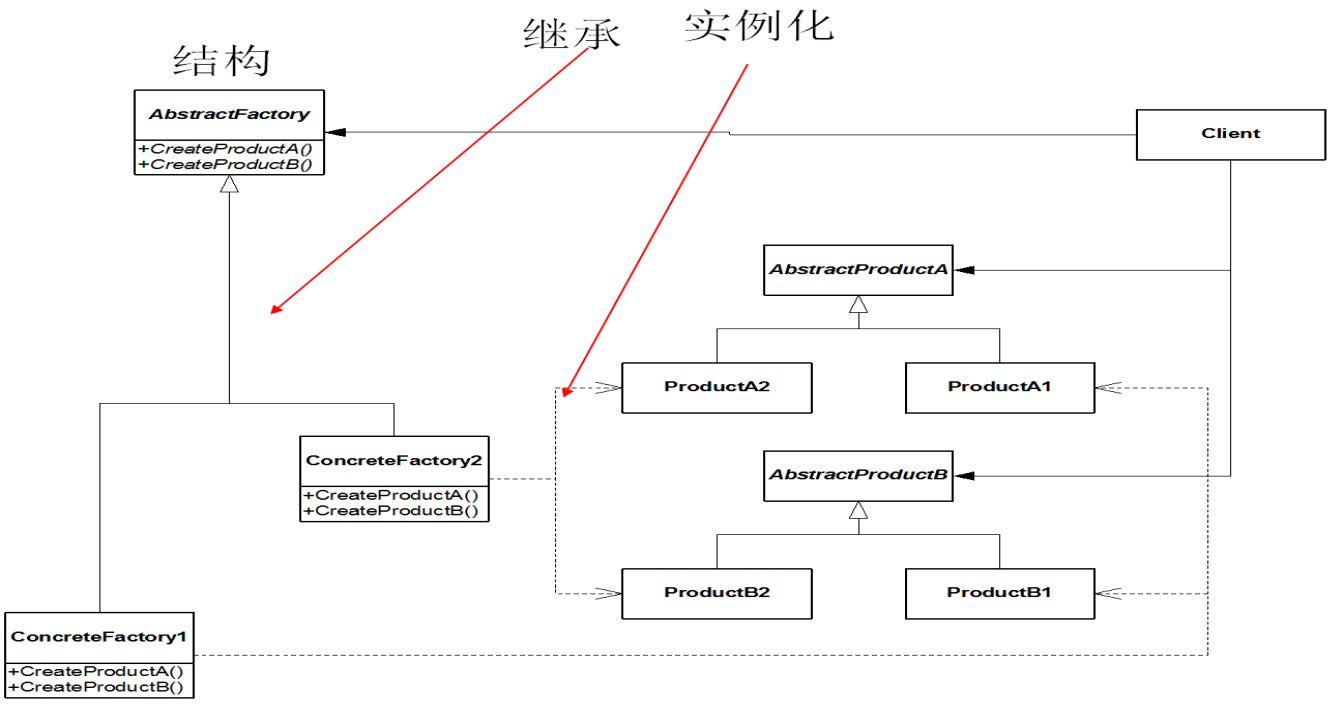


3. Factory Method（工厂方法，虚构造器）

- 意图：定义一个创建对象的接口，**让子类决定实例化哪个类**。 Factory Method 使一个类的实例化延迟到其子类。
- 例如，在铸造成型的过程中，制造者首先将塑料加工成可成型的粉末，然后将塑料粉末注入到需要的模具中。在 Factory Method 模式中，子类 (在这里就是指模具)来决定需要实例化那个类。在这个例子中，ToyHorseMold 类就是被实例化的类。

4. Abstract Factory（抽象工厂）

- 意图：提供一个创建一系列相关或者相互依赖对象的接口，不要指定它们的具体类。



- 例如， 在汽车制造中，金属片冲压设备是 *Abstract Factory* 模式的一个例子，它用来制造汽车的部件。用滚筒(rollers)来改变模具(dies)，就可以改变这样的具体对象(模具的加工品)。这些具体的对象包括：引擎罩、车的中轴、车顶、左右的前端挡板等。汽车的主控部分要确保所有这些具体的对象都是兼容的。

5. 抽象工厂模式与工厂模式的区别：

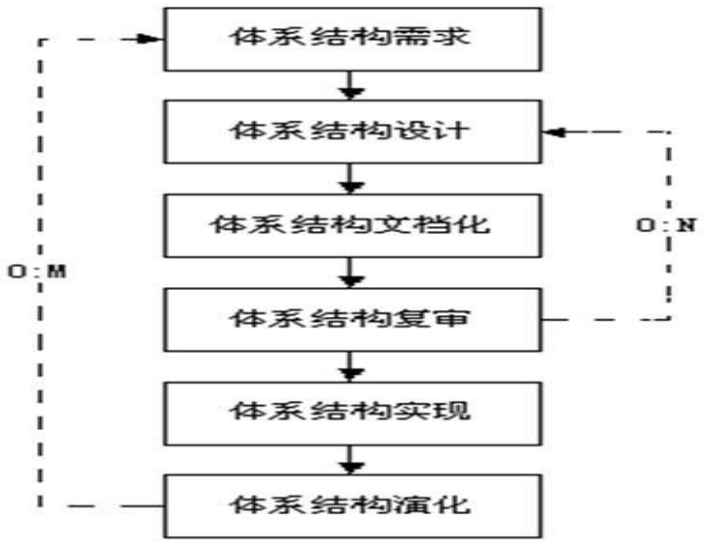
- 抽象工厂模式比工厂模式更复杂，更灵活，一个抽象工厂模式或以先创建出多个具体的工厂，这些具体的工厂再创建出具体的产品。
- 工厂要生产食用油
  - 工厂模式:只生产大豆油,产品单一;
  - 抽象工厂模式:除了大豆油,还生产色拉油,花生油,调和油……
- 抽象工厂模式关键在于工厂类是多层次的，有父工厂类和子工厂类，父工厂类可以产生子工厂类，再由子工厂类生产出产品，这样产品也可以是由复杂关系的，也可以说多种的。
- 工厂方法模式，将的是由一个方法，可以产生不同的但是同类的（或者同接口的）产品。工厂方法模式就能满足一般的需要。复杂情况下才用抽象工厂模式。
- 工厂方法采用的是类继承机制（生成一个子类，重写该工厂方法，在该方法中生产一个对象）。而抽象工厂采用的是对象组合机制，专门定义“工厂”对象来负责对象的创建。对象组合的方式就是把“工厂”对象作为参数传递。
- 工厂方法模式：一个抽象产品类，可以派生出多个具体产品类。抽象工厂模式：多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个

十三 动态体系结构含义

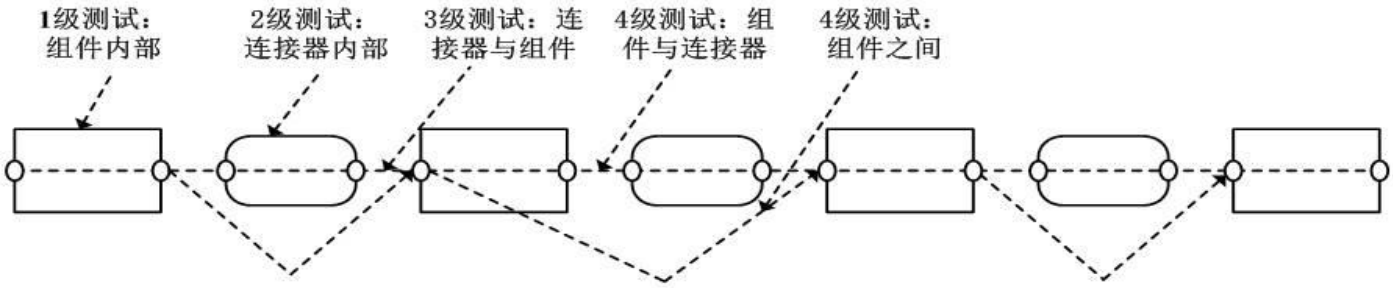
1. 软件体系结构动态性

- 演化：由于系统需求、技术、环境、等因素而导致的软件体系结构的变化。
- 动态：软件系统在运行时的体系结构的变化为体系结构的动态性。
- 静态：体系结构静态的修改，又可称为体系结构扩展

十四 ABSDM



十五 软件体系结构测试



十六 软件体系结构评估三种方法及比较

0. 性能 可靠性 可用性 安全性 可修改性 功能性 可变性 集成性 互操作性

1. 基于调查问卷或检查表的评估方式

2. 基于场景的评估方式

3. 基于度量的评估方式

比较：

评估方式	调查问卷或检查表		场景	度量
	调查问卷	检查表		
通用性	通用	特定领域	特定系统	通用或特定领域
评估者对体系结构的了解程度	粗略了解	无限制	中等了解	精确了解
实施阶段	早	中	中	中
客观性	主观	主观	较主观	较客观

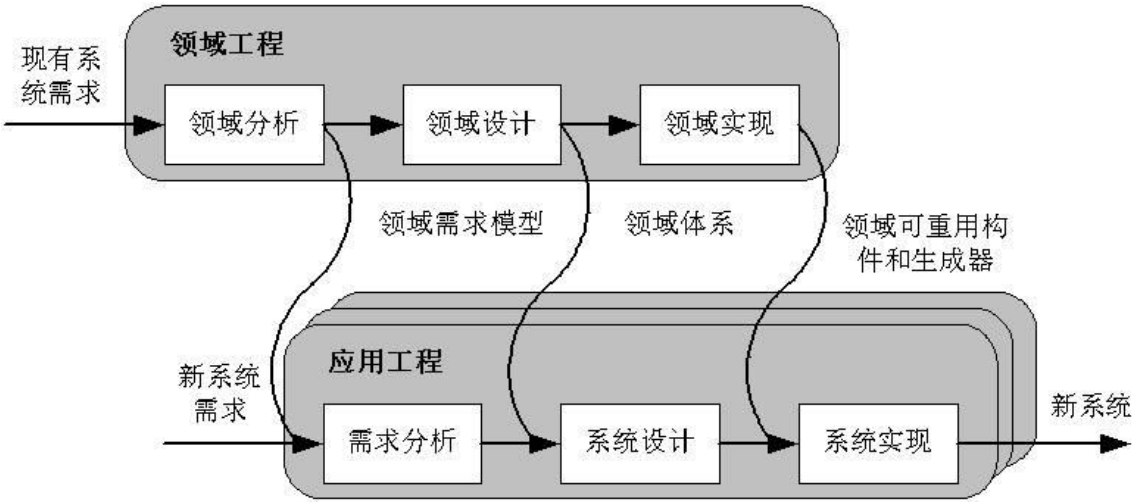
十七 产品线定义（SEI），产品线过程模型

1. 基本概念

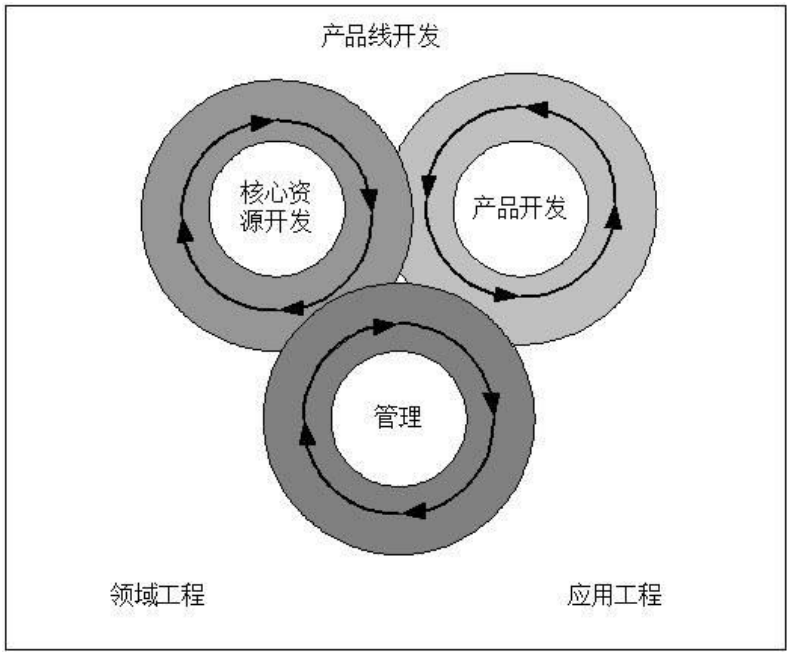
- CMU/SEI 的定义：“产品线是一个产品集合，这些产品共享一个公共的、可管理的特征集，这个特征集能满足选定的市场或任务领域的特定需求。这些系统遵循一个预描述的方式，在公共的核心资源(core assets)基础上开发的。”

2. 软件产品线的过程模型

1) 双生命周期模型



2) SEI 模型



十八 WEB 服务模型的三个构成元素以及三个基本协议

1. 定义

- Web 服务作为一种新兴的 Web 应用模式，是一种崭新的分布式计算模型，是 Web 上数据和信息集成的有效机制。
- Web 服务就像 Web 上的构件编程，开发人员通过调用 Web 应用编程接口，将 Web 服务集成进他们的应用程序，就像调用本地服务一样。

2. Web 服务的特点

- 应用的分布式
- 应用到应用的交互

- 平台无关性
- 使用标准协议规范
- 高度集成能力
- 完好的封装性
- 松散耦合

3. 三个构成元素：Service Broker，Service Provider，Service Requester

4. 三个基本协议：

- 协议一：简单对象访问协议 SOAP
- 协议二：统一描述、发现和集成协议 UDDI
- 协议三：Web 服务描述语言

5. 三个构成元素和三个基本协议的关系：

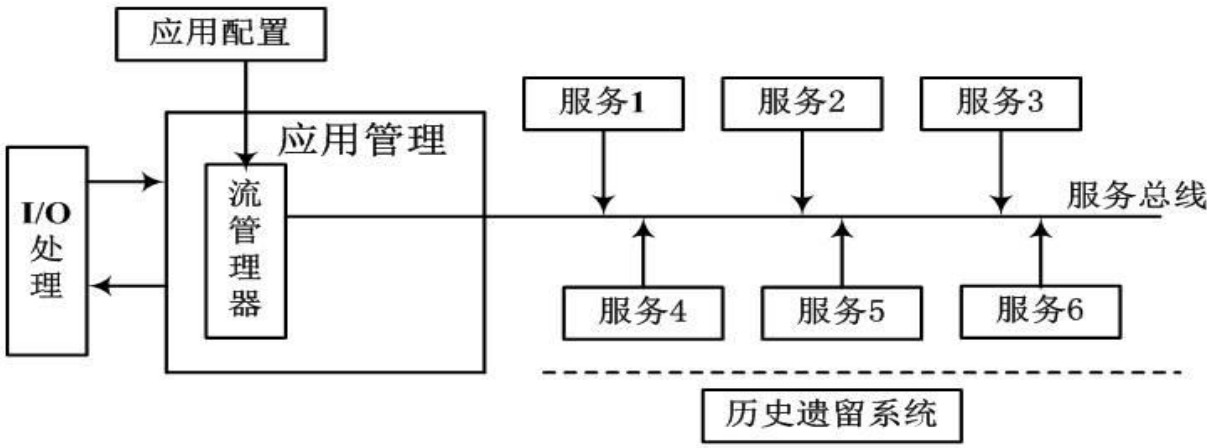
十九 SOA 及其结构

1. SOA 定义：

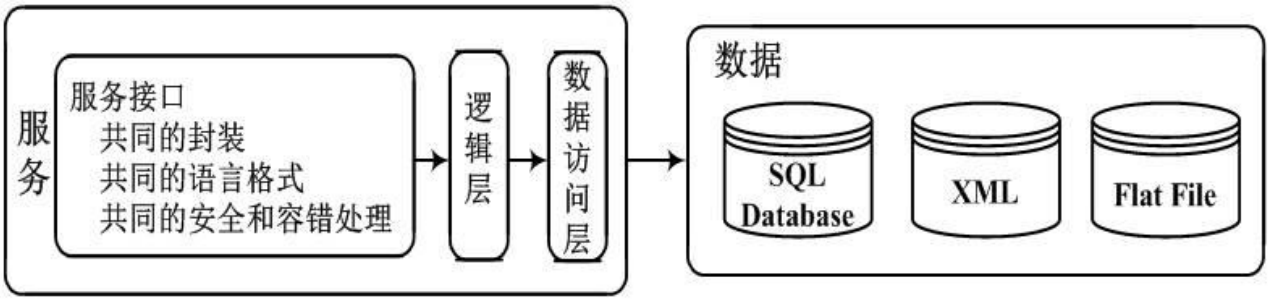
- W3C 定义
  - SOA 为一种应用程序体系结构，在这种体系结构中，所有功能都定义为独立的服务，这些服务带有定义明确的可调用接口，可以以定义好的顺序调用这些服务来形成业务流程。
- Gartner 定义
  - SOA 为客户端/服务器的软件设计方法，一项应用由软件服务和软件服务使用者组成，SOA 与大多数通用的客户端/服务器模型不同之处，在于它着重强调软件构件的松散耦合，并使用独立的标准接口。

2. SOA 结构：

- 一个完整的面向服务的体系结构模型



- 单个服务内部结构



题型：

- 名词解释/简答（大约 20）4 分/个
- 问答（大约 40）8 分/个
- 设计（大约 40）
  - 4 + 1 视图
  - ACME 语言
  - 设计模式（组合、工厂、适配器）