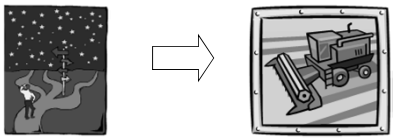
Shanghai Jiao Tong University

软件工程
Module: 设计工程

上海交通大学软件工程中心

Change our viewpoint

Do the right things ⇒ Do the things right



What How

软件需求工程解决“做什么”的问题
软件设计工程解决“怎么做”的问题

设计工程

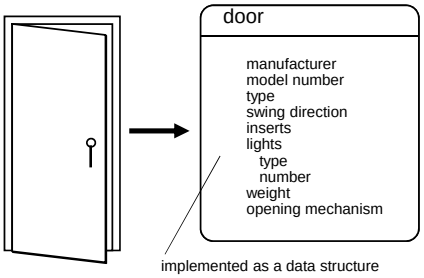
- 软件设计的原则
  - 抽象和分解 — 模块化设计
- 软件设计的步骤和方法
- 软件架构设计

@第6章.教材

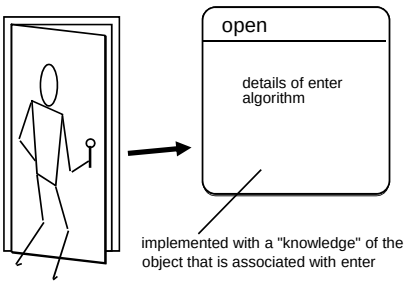
抽象 abstraction

- 抽象，是在软件规模逐渐增大、软件复杂性逐渐增大下，控制复杂性的基本策略。软件开发过程就是对软件抽象层次的一次次细化的过程：需求、架构、设计、编码。
- 抽象的好处：本质化、简单化、稳定、灵活
- 在软件设计中，主要抽象手段包括：
  - 数据抽象把一个数据对象的定义抽象为一个数据类型名，用此类型名可定义多个具有相同性质的数据对象
  - 过程抽象把完成一个特定功能的动作序列抽象为一个过程名和参数表，以后通过指定过程名和实际参数调用此过程
  - 对象抽象则通过操作和属性，组合了这两种抽象，即在抽象数据类型的定义中加入一组操作的定义，以确定在此类数据对象上可以进行的操作。

数据抽象Data Abstraction



过程抽象Procedural Abstraction



分解和模块化

- 分解（decomposition）是控制复杂性的另一种有效方法，软件设计用分解来实现模块化设计。
- 模块化（modularity），即将一个复杂的系统自顶向下地分解成若干模块（module），每个模块完成一个特性，所有的模块组装起来，成为一个整体，完成整个系统所要求的特性。

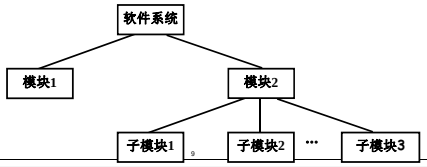

模块化是产业规模化的基础

软件模块化的好处

- 把复杂软件变简单，更易实现
- 把易变的部分和稳定的部分分开，模块可插拔可替换，应对需求变更和技术升级
  - 在计算机中，CPU是常变的模块，键盘是相对稳定的模块
- 支持多人协同开发
- 支持迭代式开发
- 模块复用和模块组装，加快开发时间，节省开发费用，提高产品质量
- 促进形成大规模开发的软件产业－软件工厂

软件模块

- 模块是能够单独命名并独立地完成一定功能的程序语句的集合
  - 例如，子系统、过程、函数、子程序、宏、类等
- 模块具有两个基本的特征：
  - 外部特征是指模块跟外部环境联系的接口和模块的功能；
  - 内部特征是指模块的内部环境具有的特点，即该模块的局部数据和处理逻辑。
- 模块可继续分解为子模块、子子模块……



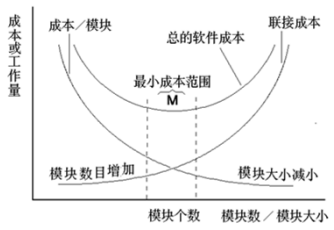
分解 decomposition

$$C(P1+P2)>C(P1)+C(P2)$$
$$E(P1+P2)>E(P1)+E(P2)$$

- C为问题的复杂度，E为解题需要的工作量
- 思考：
  - 如果我们无限制地划分软件，开发它所需的工作量会变得小到可以忽略？！
  - 事实上，影响软件开发的工作量的因素还有很多，例如模块接口费用等等
  - 上述不等式只能说明，当模块的总数增加时，单独开发各个子模块的工作量之和会有所减少

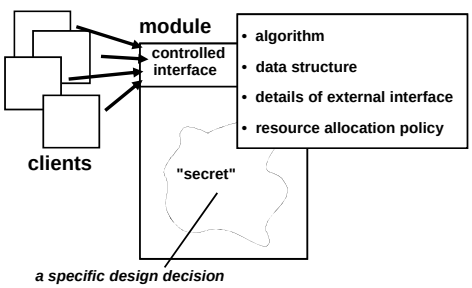
模块化的成本

- 如果模块是相互独立的，当模块变得越小，每个模块花费的工作量越低；
- 但当模块数增加时，模块间的联系也随之增加，把这些模块联接起来的工作量也随之增加。



信息隐藏 Information Hiding

- 每个模块的实现细节对于其它模块来说应该是隐蔽的



模块独立

- 模块独立的定义：
  - 模块完成独立的功能并且与其他模块的接口简单
- 模块独立的重要性：
  - 功能被划分，并且接口被简化，所以具有有效模块化的软件更易于开发。当多人分工合作开发同一个软件时，这个优点尤其重要。
  - 由于因设计和编码修改引起的副作用受到局限，错误传播被减小，并且模块复用成为可能，所以独立的模块更易于维护和测试 。

模块独立的衡量指标

- 模块独立可以由两项指标来衡量：
  - 内聚（cohesion），是一个模块内部各个元素彼此结合的紧密程度的度量
  - 耦合（coupling），是模块之间的相对独立性（互相连接的紧密程度）的度量
- 模块独立追求高内聚和低耦合。

内聚

- 一般模块的内聚性分为七种类型：
  1. 偶然内聚 coincidental cohesion
  2. 逻辑内聚 logical cohesion
  3. 时间内聚 temporal cohesion
  4. 过程内聚 procedural cohesion
  5. 通信内聚 communicational cohesion
  6. 顺序内聚 sequential cohesion
  7. 功能内聚 functional cohesion

低 → 高

- 确定内聚的精确级别是不必要的，重要的该是尽量争取高内聚和识别低内聚

低、中、高内聚

- 低内聚：
  - 有时在写完一个程序之后，发现一组语句在两处或多处出现，于是把这些语句作为一个模块，这样就出现了偶然内聚的模块。
  - 如果一个模块完成的任务在逻辑上属于相同或相似的一类（例如，一个模块产生各种类型的全部输出），则称为逻辑内聚。
  - 如果一个模块包含的任务必须在同一段时间内执行（例如，模块完成各种初始化工作），就叫时间内聚。
- 中内聚：
  - 如果一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚。
  - 如果模块中所有元素都使用同一个输入数据和（或）产生同一个输出数据，则称为通信内聚。
- 高内聚：
  - 如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行，则称为顺序内聚。
  - 如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚。功能内聚是最高程度的内聚。

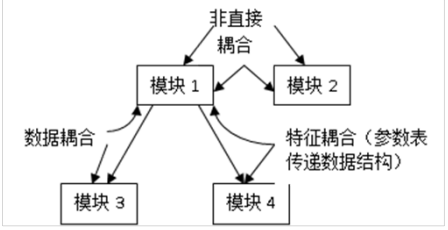
耦合

- 一般模块之间可能的耦合方式有七种类型
  1. 非直接耦合 no direct coupling
  2. 数据耦合 data coupling
  3. 特征耦合 stamp coupling
  4. 控制耦合 control coupling
  5. 外部耦合 external coupling
  6. 公共耦合 common coupling
  7. 内容耦合 content coupling

低 → 高

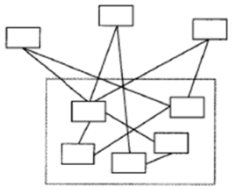
- 确定耦合的精确级别是不必要的，重要的该是尽量争取低耦合和识别高耦合

弱耦合示例

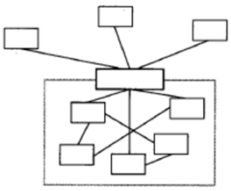


- 模块1与模块2为同级模块，相互之间没有信息传递，属于非直接耦合。
- 模块1调用模块3，交换的是简单变量，便构成数据耦合；
- 模块1调用模块4，交换的是数据结构，则构成特征耦合。

哪个模块化设计好？



设计A



设计B

设计工程

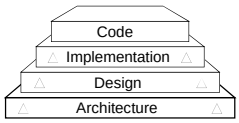
- 软件设计的原则
- 软件设计的步骤和方法
- 软件架构设计

设计的步骤

- 架构设计 (Architecture Design)
  - 又称概要设计，定义了软件的全貌(即蓝图)，记录了最重要的设计决策，并成为随后的详细设计与实现工作的战略指导原则。
  - 独立于面向对象、结构化等方法进行设计，可采用AADL等可视化建模语言进行刻画。
- 详细设计 (Detail Design)
  - 又称构件级设计，在软件架构的基础上定义各模块的内部细节，其所做的设计决策常常只影响单个模块的实现。
  - 面向对象设计、或结构化设计、或面向服务设计等

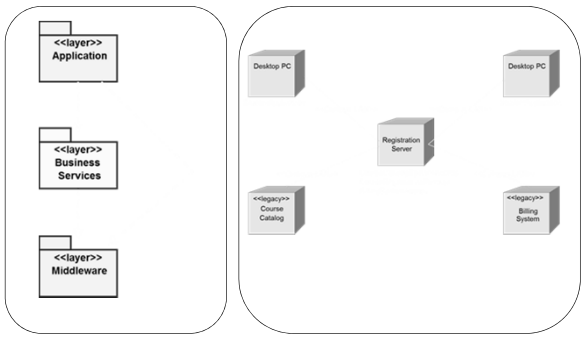
架构设计

- Architecture involves a set of strategic design decisions, rules or patterns that constrain design and construction.



Architecture decisions are the most fundamental decisions, and changing them will have significant effects.

选课系统的架构设计

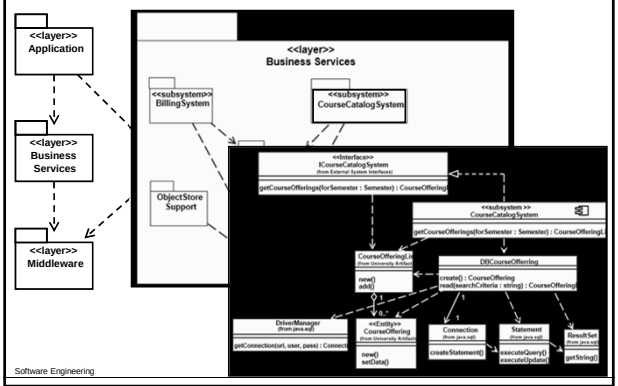


逻辑架构　　物理架构

详细设计

- 详细设计软件架构中的每个模块：
  - 将模块进一步细分：子模块－子子模块－***
    - 原子模块为类、函数、过程等
  - 为所有数据对象定义详细的数据结构
  - 为所有在模块内发生的处理定义算法细节、控制流和数据流

选课系统的详细设计



软件设计的质量要求

1) 设计应当模块化，高内聚、低耦合。
   - 支持多人合作开发
   - 易于测试和修改
   - 能够以演化过程实现
2) 设计应当包含数据、体系结构、接口和构件的清楚的表示。
3) 设计应根据软件需求采用可重复使用的方法进行。
4) 应使用能够有效传达其意义的表示法来表达设计模型。

7种软件设计的坏味道

1）僵化性（Rigidity）
- 很难对软件进行改动，因为每个改动都会迫使对系统其他部分的许多改动

2）脆弱性（Fragility）
- 对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题

3）牢固性（Immobility）
- 很难解开系统中某部分与其它部分之间的纠结，从而难以使其中的任何部分可以被分离出来被其它系统复用

4）粘滞性（Viscosity）
- 做正确的事情要比做错误的事情困难。表现为两种形式：
  - 软件粘滞性
    - 需要对软件进行修改时，可能存在多种方法。有的方法可以保持原有的设计质量，另一些方法则会破坏原有的设计质量。如果，破坏软件质量的修改比保持原有设计质量的修改更容易实施时，我们就称该软件具有“软件粘滞性”。
  - 环境粘滞性
    - 当开发环境迟钝、低效时，就会产生环境粘滞性。
    - 例如：如果编译时间很长，那么开发人员可能会放弃那些能保持设计质量，但是却需要导致大规模重新编译的改动。

5）不必要的复杂性（Needless Complexity）
- 设计中包含不具有任何好处的基础结构。

6）不必要的重复（Needless Repetition）
- 设计中包含一些重复的结构，这些结构本来可以通过单一的抽象进行统一
  - 使用Cut/Copy/Paste实施源代码级的软件复用容易导致这一问题
  - 这种代码级别的冗余，将带来修改上的问题

7）晦涩性（Opacity）
- 很难阅读和理解，不要相信你永远都会如此清楚的了解你的每一行代码，“时间会冲淡一切”。要站在阅读者的角度进行设计

如何进行高质量的设计？

复用现有好的设计方案
——模式(Pattern)的复用！

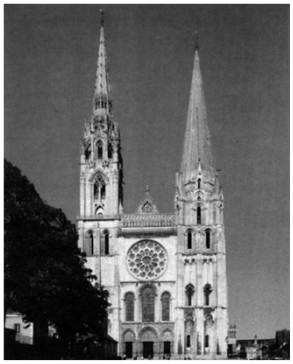
- Christopher Alexander 说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”，1977年。
- 定义:“在一个上下文中对一种问题的解决方案”。

建筑模式

古希腊、古罗马建筑

古罗马大斗兽场

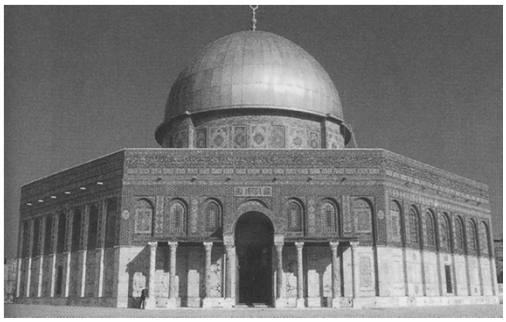
建筑模式——哥特建筑



法国夏特尔主教堂

建筑模式——中国建筑

建筑模式——伊斯兰的清真寺



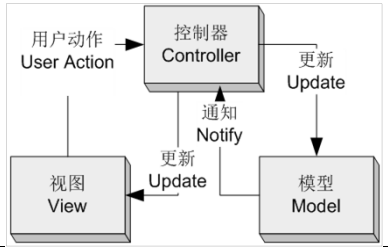
耶路撒冷的圣石庙

软件设计的模式分类

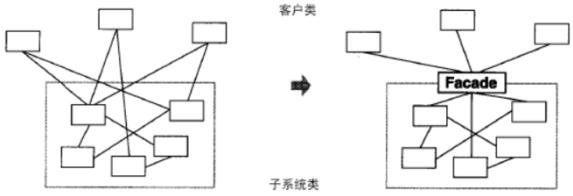
- 架构风格Architectural style
  - 例如分层架构风格、MVC风格
- 设计模式Design pattern
  - 例如Facade模式、工厂模式、单例模式
- 编程惯用Idiom
  - 例如Java 多线程编程模式

架构风格举例：MVC

模型Model：管理系统中存储的数据和业务规则，并执行相应的计算功能。
视图View：根据模型生成提供给用户的交互界面，不同的视图可以对相同的数据产生不同的界面。
控制器Control：接收用户输入，通过调用模型获得响应，并通知视图进行用户界面的更新。



设计模式举例：Facade模式



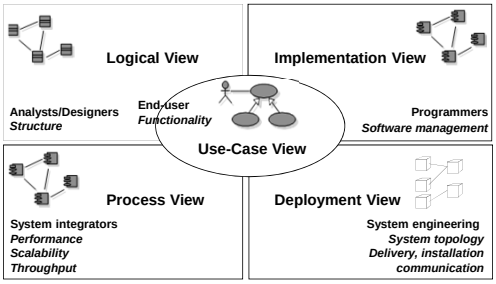
设计工程

- 软件设计的原则
- 软件设计的步骤和方法
- 软件架构设计

软件架构设计 (software architecture design)

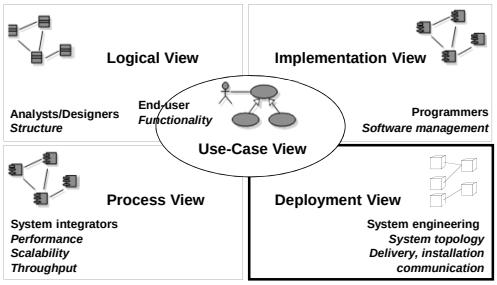
- 架构设计的内容：
  - ① 设计软件架构的多个视图
  - ② 选择软件质量属性（性能、可靠性、安全性、可移植性、可扩展性等）的设计策略
- 目标：使得软件系统在架构层面的设计上满足拟建系统功能性和非功能性需求。

① 设计软件架构的多个视图



Software Architecture: The “4+1 View” Model

部署视图（物理视图）



The Deployment View is an “architecturally significant” slice of the Deployment Model.

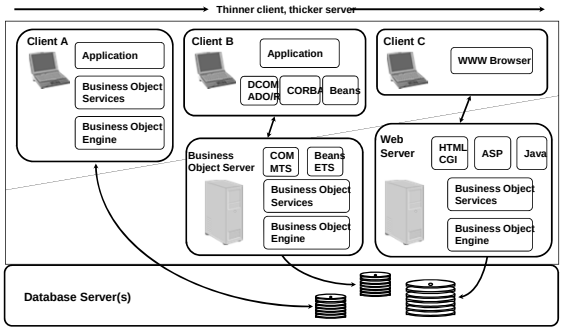
架构的部署视图

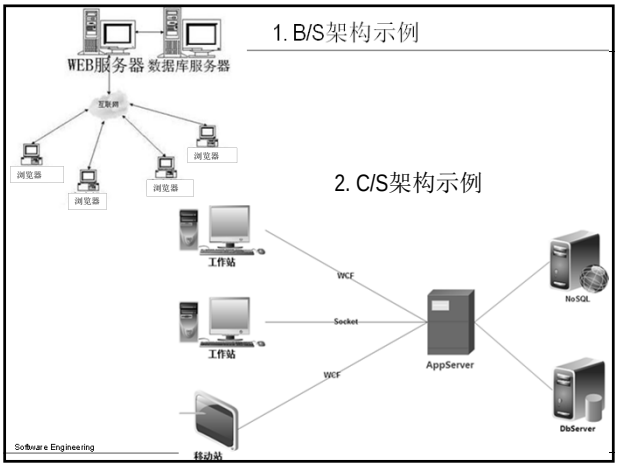
- 针对分布式系统，需要定义架构的部署视图，刻画计算机或处理节点以及网络间的关系。
- 单机软件不需要定义部署视图。
- 重点考虑可靠性、性能、可伸缩性等非功能需求的支持。

部署架构风格

- Client/Server
  - 3-tier
  - Fat Client
  - Fat Server
  - Distributed Client/Server
- Peer-to-peer (P2P)

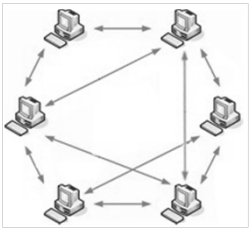
Client/Server Architectures





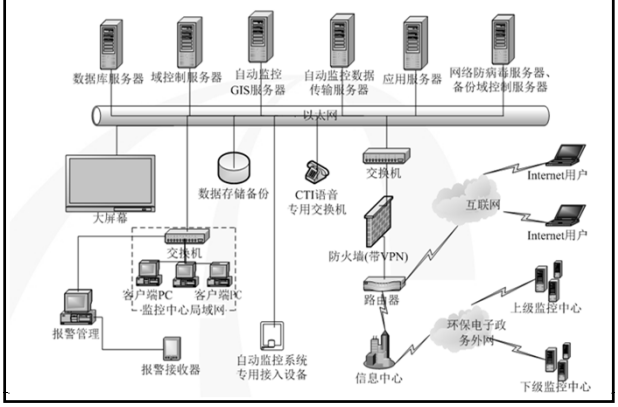
P2P物理架构

- 所有计算机节点都是对等的

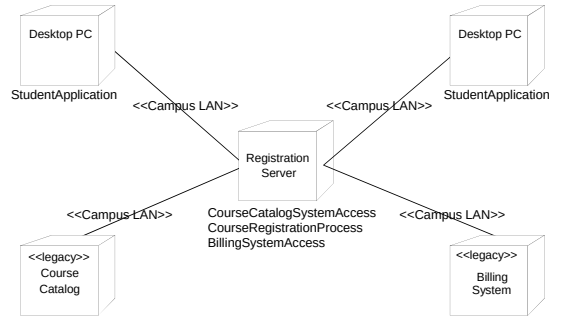


- 讨论：网上四人飞行棋软件的物理架构？

举例：环境远程监控系统的物理架构

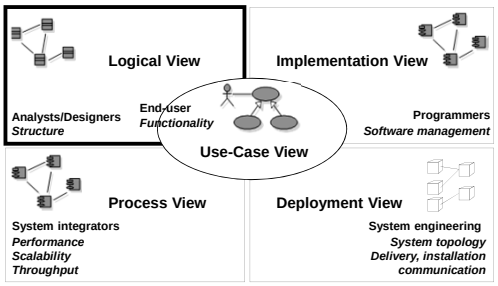


UML的物理架构的描述



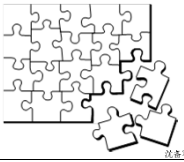
选课系统的C/S物理架构

逻辑视图

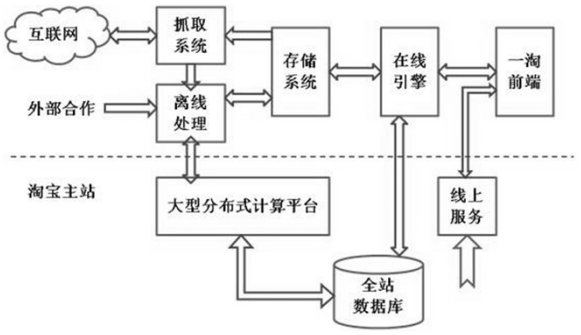


架构的逻辑视图

- 把软件划分为多个模块（又称构件），模块和模块相互协作，共同完成软件的需求规约。这些模块将部署在物理架构的同一节点或不同节点上。
- 重点考虑功能、可维护性等需求的支持。
- 所有软件都需要定义逻辑架构



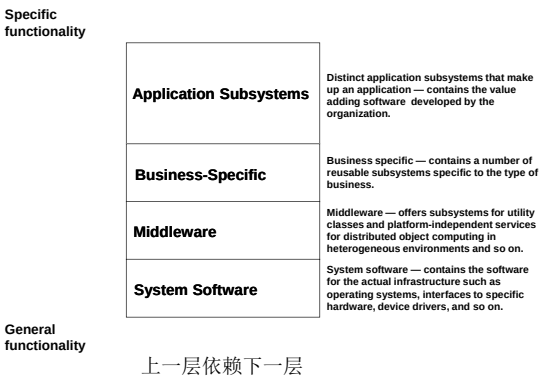
举例：一淘网（www.etao.com）的逻辑架构图



逻辑架构风格

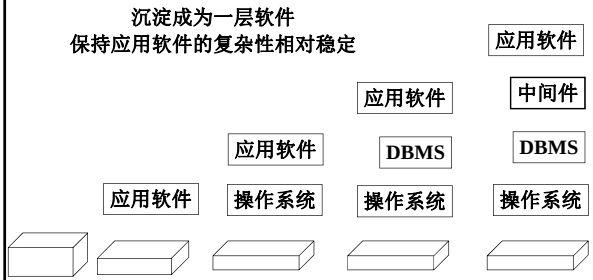
- 表现层分离风格：MVC
- 数据流风格 (Dataflow)：批处理序列、管道－过滤器风格 (Pipe-and-Filter)
- 调用/返回风格：主程序/子程序、面向对象风格 (ADT)、多层 (Layer)
- 分布计算风格：多层（Tier)、代理、C/S、P2P
- 独立构件风格：事件响应、消息总线、微服务
- 虚拟机风格：解释器、基于规则的系统
- 仓库风格：数据库系统、超文本系统、黑板系统
- 自适应风格：微内核、反射、控制反馈
- ……

1) 层次架构（layered architecture）风格



上一层依赖下一层

中间件



中间件的分类

1) 数据访问中间件
- 允许应用程序和本地或者异地的数据库进行通信，并提供一系列的应用程序接口（如ODBC、JDBC等）。该类中间件技术上最成熟，但局限于与数据库相关的应用。

2) 消息中间件
- 可以屏蔽平台和协议上的差异进行远程通信，实现应用程序之间的协同，如IBM的MQSeries、BEA的MessageQ、SUN 的JMS和微软的MSMQ 等，其优点在于提供高可靠的同步和异步通信，缺点在于不同的消息中间件产品之间不能互操作。

3) 远程过程调用RPC中间件
- 解决了平台异构的问题，但编程复杂且不支持异步操作。

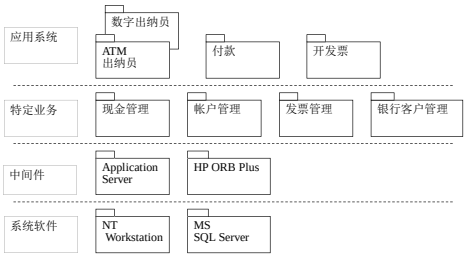
4）事务中间件
- 是在分布、异构环境下提供保证事务完整性和数据完整性的一种平台，如BEA的TUXEDO、IBM的CICS、微软的MTS 。其优势在于对关键业务的支持，但机制复杂、对用户要求较高。

中间件的分类 （2）

5）分布对象中间件
- 在分布、异构的网络计算环境中，可以将各种分布对象有机地结合在一起，完成系统的快速集成。主流标准有Microsoft的DNA/COM+、OMG的OMA/CORBA、Sun的 J2EE / EJB。Weblogic, Websphere, Jboss, .Net等应用服务器都包含了分布对象中间件，也有如Orbix、HP ORB等独立产品。

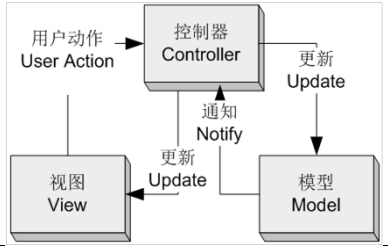
6）Web服务中间件
- 根据需求通过网络对松散耦合的粗粒度应用服务进行分布式部署、组合和使用，其标准是SOA，Web服务是其中的一种实现。应用服务器都包含了Web服务中间件，也有AXIS2、HP Web Services Platform等独立产品。

层次架构实例

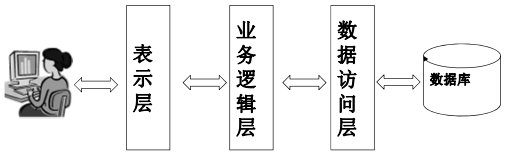


2) MVC

模型Model：管理系统中存储的数据和业务规则，并执行相应的计算功能。
视图View：根据模型生成提供给用户的交互界面，不同的视图可以对相同的数据产生不同的界面。
控制器Control：接收用户输入，通过调用模型获得响应，并通知视图进行用户界面的更新。

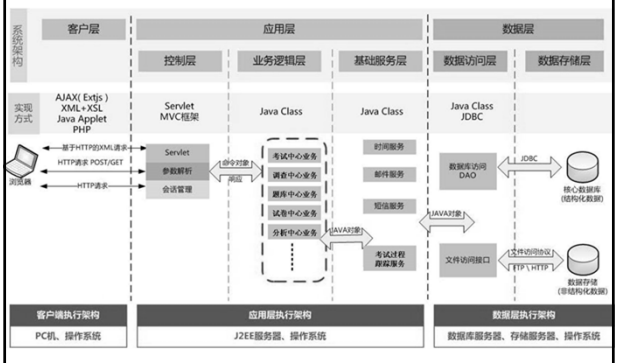


3) 3 Tiers



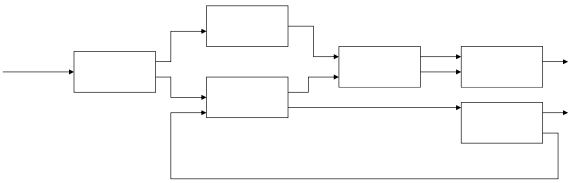
- 表示层：负责向用户呈现界面，并接收用户请求发送给业务逻辑层；
- 业务逻辑层：负责执行业务逻辑以处理用户请求，并调用数据访问层提供的持久性操作；
- 数据访问层：负责执行数据库持久性操作。

举例：基于Web的在线考试系统的架构图



4) 管道和过滤器（Pipes and Filters）

- In this style, each component has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs and produces streams of data on its output.

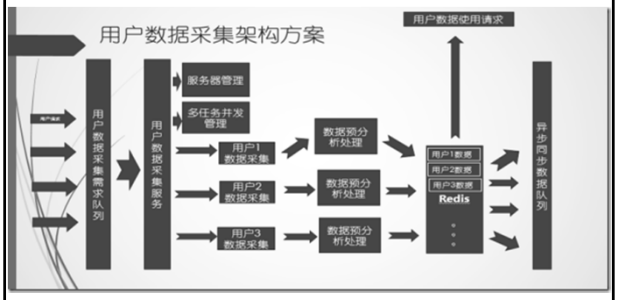


举例

- Linux的Shell程序可以看做是典型的管道与过滤器架构的例子
- 例如下面的Shell脚本：
  - $cat *TestResults* | sort | grep *Good*

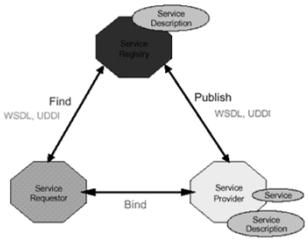
会将TestResults文件的文本进行排序，然后找出其中包含单词Good的行，并显出在屏幕上。Shell命令cat、sort和grep依次执行，就构成了一个管道-过滤器架构。

举例



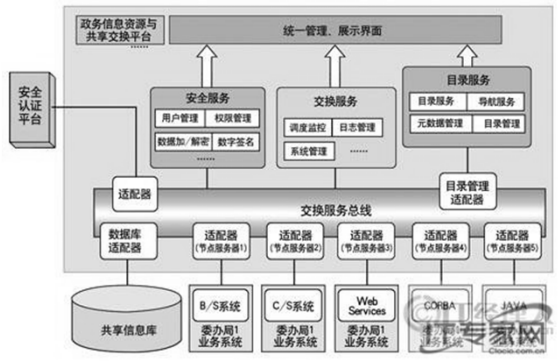
5）服务与微服务的架构风格

服务架构风格



- 服务的抽象性（基于接口的编程）
- 服务的自治性（实现分布式应用）
- 服务间的松耦合式绑定，基于消息进行通信
- 服务的粗粒度
- 由BPEL或ESB总控

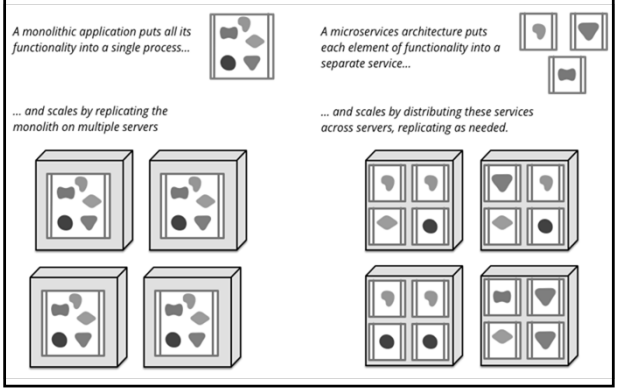
举例



微服务架构风格

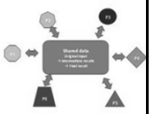
- **微服务架构风格**是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，通过轻量的通讯机制联系，经常是基于HTTP资源API，这些服务基于业务能力构建，能够通过自动化部署方式独立部署，这些服务自己有一些小型集中化管理，可以是使用不同的编程语言编写。
- **和SOA不同**：SOA倡导粗粒度服务，而它是细粒度服务。同时，微服务采用"智能终端和哑管道"，它们拥有自己的领域逻辑，以类似Unix管道过滤方式运行，接受到一个请求，使用相应的逻辑，产生一个响应，这些都可以使用RESTful方式编排，而不是使用复杂的协议如WS-Choreography 或 BPEL或ESB指挥控制。

Monolithic application Vs. Microservices



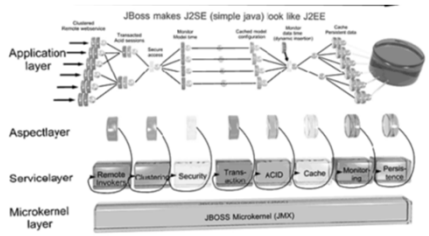
6) 仓库风格

- 仓库风格是以数据为中心的系统架构，它细分为：
  - 数据库系统
    - 以数据库为核心，各个构件存取数据。
  - 超文本系统
    - 用超链接的方法，将各种不同空间的文字、图片等信息组织在一起的网状文本
  - 黑板系统
    - 为参与问题解决的知识源提供了共享的数据表示，这些数据表示是与应用相关的。在黑板系统中，控制流是由黑板数据的状态决定的，而并非按照某个固定的顺序执行。

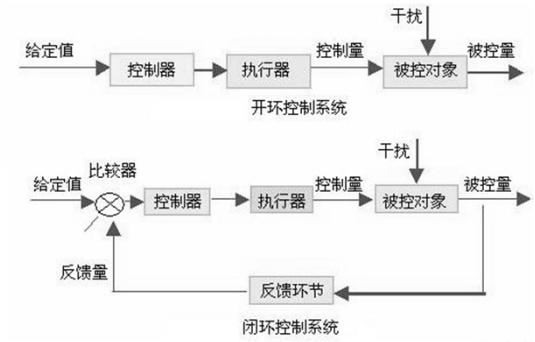


7) 微内核风格

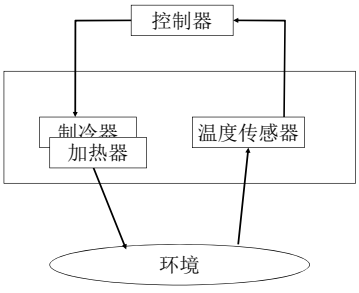
- 微内核概念来源与操作系统领域。微内核是提供了操作系统核心功能的内核，它只需占用很小的内存空间即可启动，并向用户提供了标准接口，以使用户能够按照模块化的方式扩展其功能。现在大多数操作系统都采用了微内核架构。



8) 开环和闭环控制风格

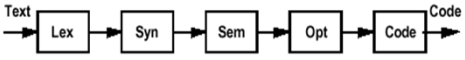


举例: 空调控制软件的逻辑架构图

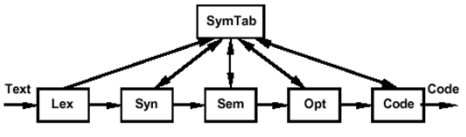


综合举例：编译器的逻辑架构图

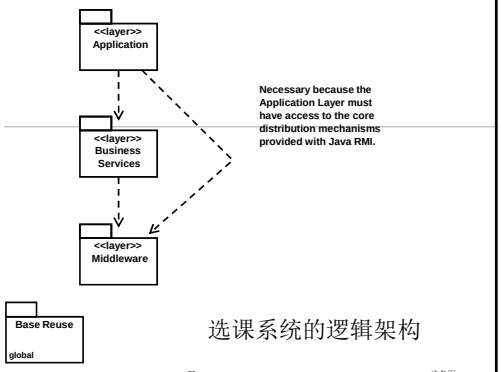
(1) 传统的编译器的架构图



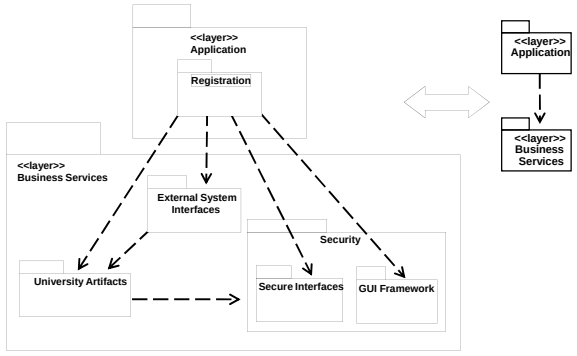
(2) 具有共享符号表的编译器的架构图



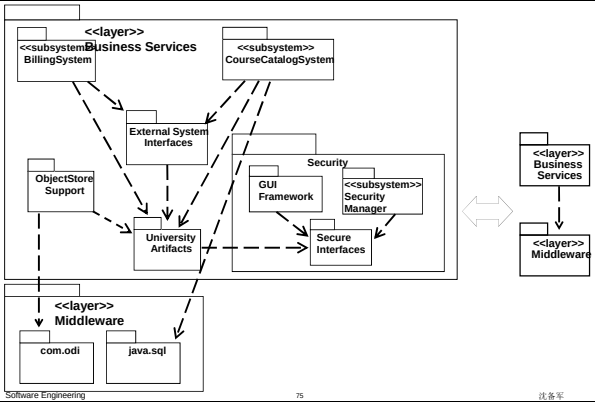
UML的逻辑视图的描述



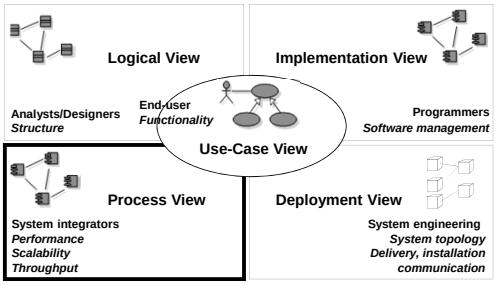
Example: Application Layer



Example: Business Services Layer Context



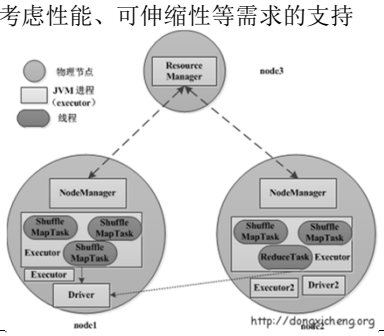
进程视图



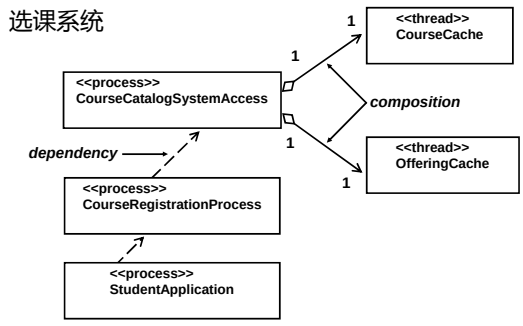
The Process View is an "architecturally significant" slice of the processes and threads of the Design Model

进程视图

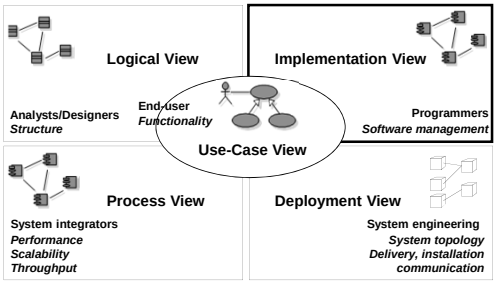
- 概念：进程(process) 与线程(thread)
- 针对多进程多线程的软件进行建模
- 重点考虑性能、可伸缩性等需求的支持



UML的进程视图的描述



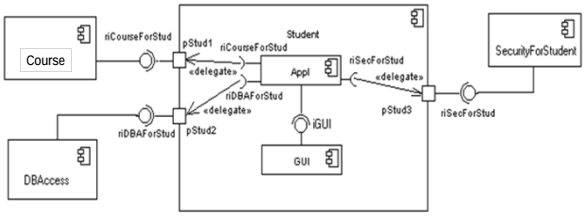
实现视图



The Deployment View is an "architecturally significant" slice of the Deployment Model.

构件图Component Diagram

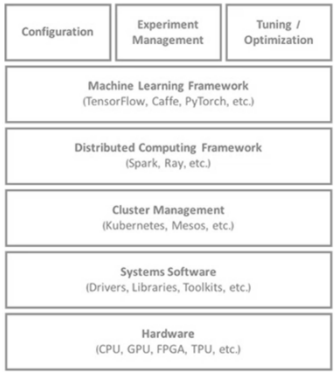
- A diagram that shows the organizations and dependencies among components
- 针对中大型软件进行建模



How many views?

- Simplified models to fit the context
- Not all systems require all views:
  - Single processor: drop deployment view
  - Single process: drop process view
  - Very Small program: drop implementation view
- Adding views:
  - Technical view, Data view, Page view, Security view .....

技术视图 Technical view

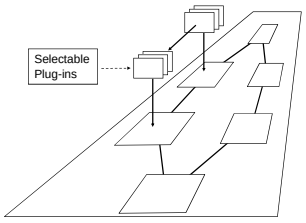


技术选型:

- 编程语言
- 操作系统
- 数据库
- 框架
- 中间件
- 库
- ....

框架(Framework)

- 框架是一个代码骨架，可以使用为解决问题而设计的特定类或功能来填充这个代码骨架，使之丰满。
- 使用框架后，该框架实现的内容就不需要再进行设计与实现



Frameworks with Plug-ins

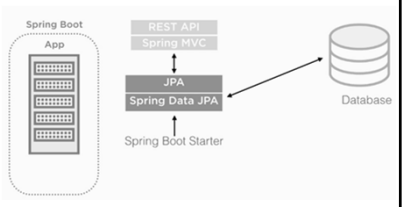
框架举例

1. MVC框架
   - Spring MVC、Struts 2、JSF、Grails、Google Web Toolkit（GWT）
2. ORM 框架
   - Mybatis
   - Hibernate、Spring Data JPA
3. Web前端框架
   - CSS框架：Bootstrap，Fundation
   - JS框架：VUE.JS, React.js, AngularJS, Ember.js
4. 并行计算框架
   - Hadoop, Spark, MapReduce
5. 服务框架
   - JAX-RS1.0+Jersey/CXF，Spring，Dubbo

等等

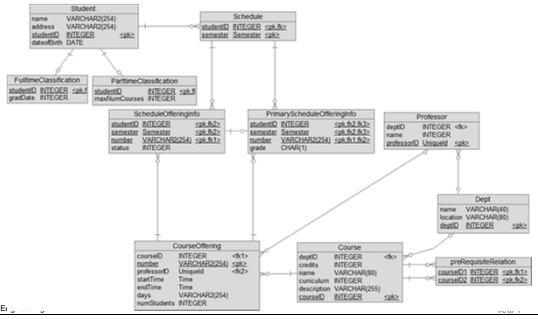
Web应用的框架推荐

- 后台框架
  - Java用Spring
  - javascript用node.js
  - python用Django
- 前台框架
  - React
  - Vue
  - Angular JS



数据视图Data View

- 针对以数据为中心的软件，则需定义数据架构
- 举例：选课系统的数据视图，从概念模型生成并进行了优化。



O-R Mapping

- The Problem:
  - As with relational databases, a representation mismatch exists between objects and these non-object-oriented formats.
- The Solution:
  - O-R Mapping service
    - a persistence service translate objects into records and save them in a database,
    - and translate records into objects when retrieving from a database
- O-R Mapping Middleware or Persistence Framework
  - Such as Hibernate, Spring Data JPA, MyBatis

②选择软件质量属性的设计战术

质量因素	设计战术
易用性	1.为用户提供适当的反馈和协助 2.将用户接口与应用的其余部分分离 3.提供“取消”、“撤消”等命令 4.建立用户模型、任务模型和系统模型

- 案例讨论：用户纷纷反馈12306网站的登录的成功率低、易用性差，你有什么改进办法？

可靠性的设计战术

质量因素	设计战术
可用性 即可靠性	1.错误检测，如心跳、异常、命令/响应 2.错误恢复，如表决、冗余、备件、检查点/回滚 3.错误预防，如事务、进程监视、从服务中删除

- 案例讨论：车辆拍牌系统在牌照抢拍时多次宕机，如何提高系统的可靠性？

性能的设计战术

质量因素	设计战术
性能	1.资源需求，如提高计算效率、减少计算开销、控制采样频率、限制队列大小 2.资源管理，如引入并发、增加可用资源、维持数据或计算的多个副本 3.资源仲裁，如先进先出、优先级调度

- 案例讨论：淘宝网站在11.11面临大量并发用户和订单，如何保证系统的性能（响应时间）？如何设计按需伸缩的软件架构？

安全性的设计战术

质量因素	设计战术
安全性 Security	1.抵抗和检测攻击，如用户身份验证、用户授权、限制暴露的信息、防火墙 2.从攻击中恢复，如维持审计追踪、采用可用性中恢复战术

- 案例讨论：如何保证银行系统的安全性？
  可以应用哪些技术与措施？

可测试性与可维护性的设计战术

质量因素	设计战术
可测试性	1.测试的输入/输出，如记录/回放、将接口与实现分离、特化访问路线/接口 2.内部监视，如内置监视器
可维护性	1.局部化修改，如泛化模块、预期期望的变更、限制可能的选择 2.防止连锁反应，如信息隐藏、维持现有接口、限制通信路径、仲裁者的使用 3.推迟绑定时间

举例：大型电商网站的物理架构