

# 软件工程

大连理工大学软件学院

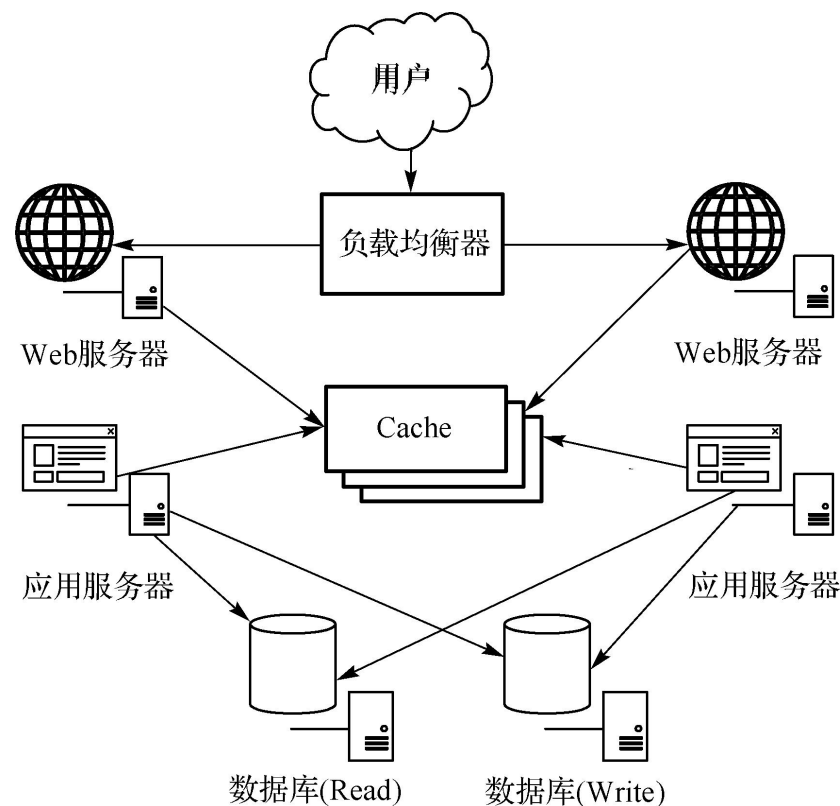


# 第9章 实现技术

- 实现阶段的一个首要任务就是构造合适的程序架构以使得所有的用户需求能够在未来的系统中得到满足和体现。
- 从设计到代码的转换过程不仅具有针对算法过程的实现，同时还要考虑到每个具体项目的约束条件。
- 与实现相关的关键技术：数据管理策略和方法、数据持久化、XML、领域特定语言(DSL)、模型驱动架构(MDA)、重构(Refactoring)等。

# 非功能性需求的实现

- 由于非功能性需求涉及的范围广且类型不尽相同，因此需要在设计和实现中根据不同的要求区别对待。
- 硬件是提升性能的手段之一，但算法、资源利用情况等也需考虑和监控（必要时）。
- 质量方面（正确性）的需求：可测试性、程序结构。
- 安全方面的要求：传输安全性、数据安全性、操作安全性。（入侵与健壮性）



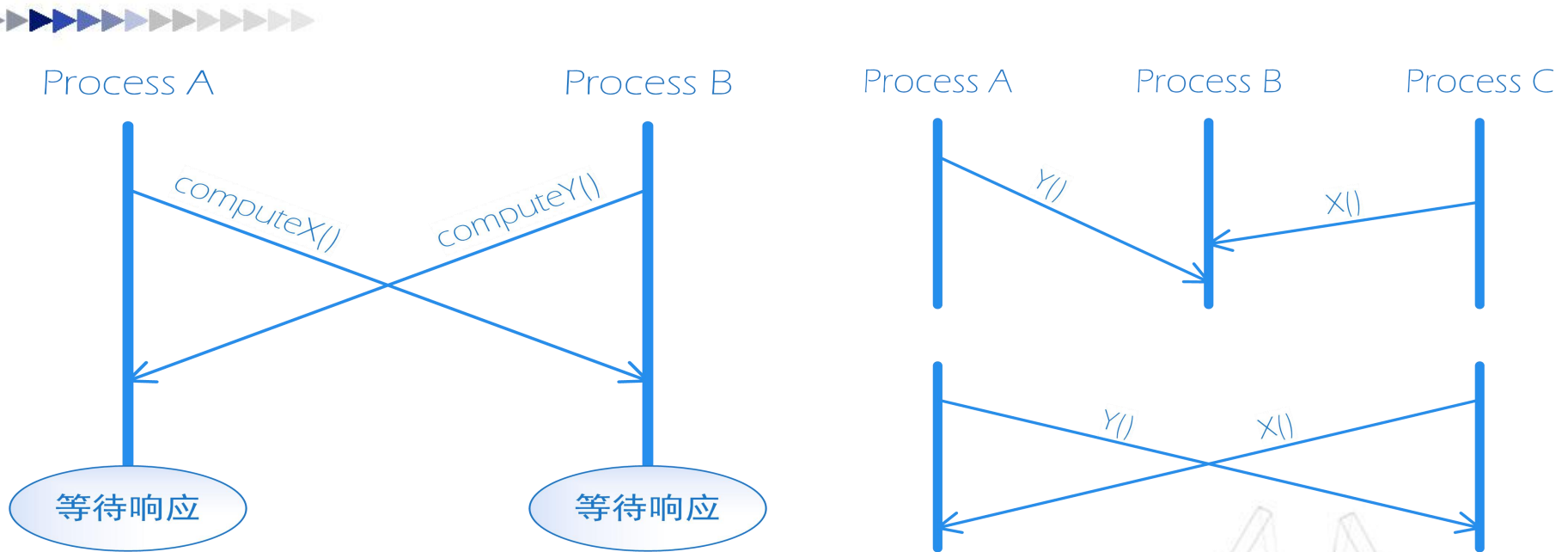
# 分布式系统

- 分布系统中存在多个控制点，因为有多个子程序需要同时工作，如操作系统中的进程。
- 进程间信息交换的方式和规则是信息领域中一个较为复杂的问题，解决的方法与条件要求（关键区域）密切相关，如数据库的事务。
- 实现要求：同构与异构、对象在网络中的分布管理、同步调用与异步调用等。

# 同步与异步调用

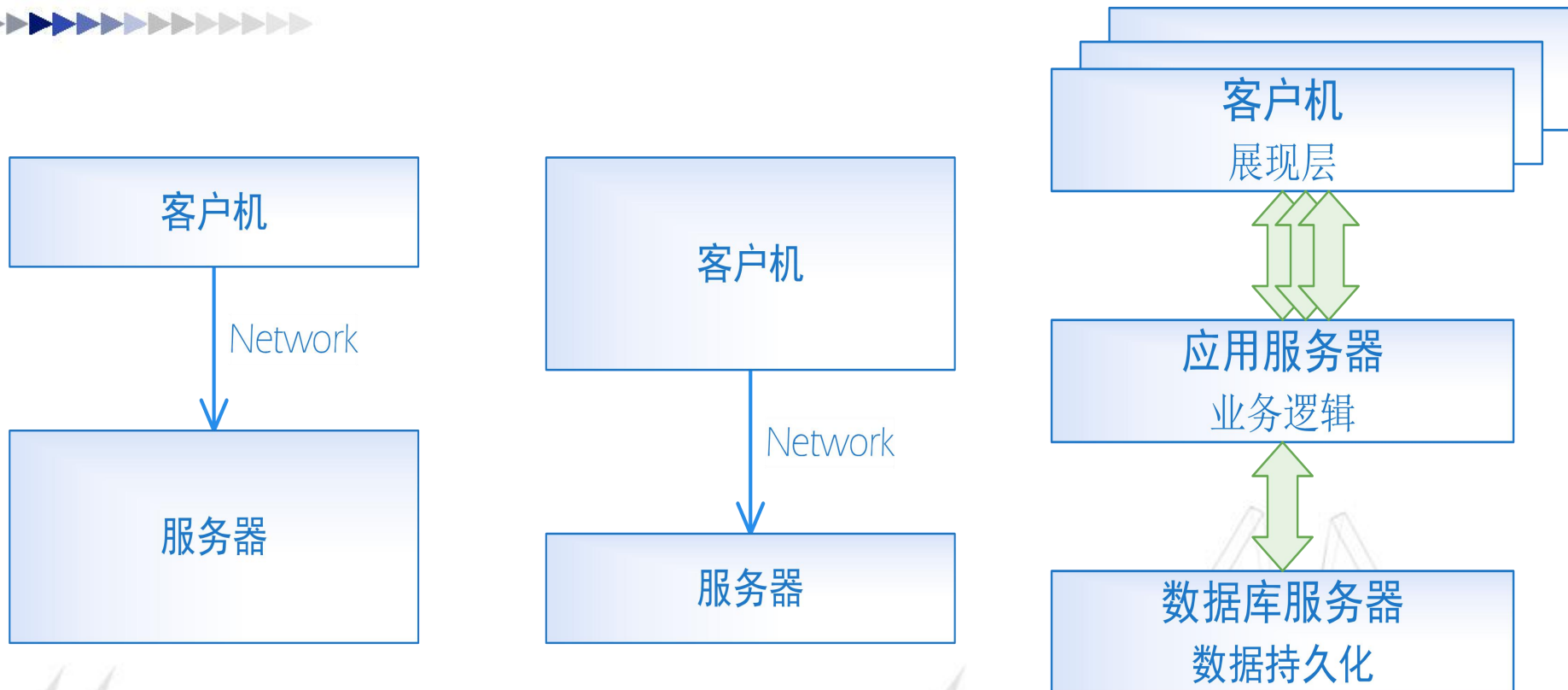
- 同步调用具有的最大优点是所有的进程相互了解各自在通信过程中所处的状态
- 同步调用缺点是需要实现相对复杂的同步通信，由于发送方和接收方需要相互等待会使得两个进程在总体上的执行速度变慢。
- 异步调用的执行速度通常是比较快的，因为发送方和接收方可以互相独立的工作。
- 异步调用容易出问题的地方是当缓冲区满的情况，这时整个系统运行变慢或者信息可生丢失。

# 死锁、竞争与活锁



- 死锁的产生、进程的竞争
- 活锁又被称为饿死（starvation）
- 解决的方法：调度问题，在公平性和合理性上做出折衷。

# 胖客户、瘦客户、三层架构



- 三层架构实现了视图、业务和数据的分离，提升系统可维护性的同时也为系统分布提供更好的选择。

# XML



- XML是由W3C委员会定义的一种标准化语言，用来描述数据模型和数据。

```
<project department="Development" contract="Fixedprice" >
<projectname>Storage Module</projectname>
<customer bankno="75566445" accountno="35634534" />
<projectleader cost="1000" >
<empno>49</empno>
<name>Udo Kelter</name>
<deptno>50</deptno>
</projectleader>
</customer>
</project>
```

- XML结构的描述：DTD（Data Type Definition）和XSD（XML Schema Definition）。
- 如果XML文档对其标准的DTD或XSD进行了说明并进行了约束，则称该文档为well formed，指定了该XML文档的有效性。





# XML程序设计

- 对于XML文档的存储和处理目前已经出现了很多可用的软件包或系统，使得每种编程语言都具有了处理XML文档的能力。
- XML的处理方式一般有两种：文档对象模型（DOM）或用于XML的简单API（SAX）。
- DOM是复杂对象处理的首选，比如当XML比较复杂的时候，或者当需要随机处理文档中数据的时候。
- SAX则是以流的方式从文档的开始通过每一节点进行移动，以定位一个特定的节点。

# 程序轮子

- 对于大多数经常出现的问题，可以将常见的解决方法通过库函数的形式提取出来作为一种公共的资源共享。
- 库函数可以是免费或者收费的，应尽量选取那些使用者较多的函数库，因为存在的缺陷可能也要少些。
- 在对应用进行测试时，通常是在假设函数库正确的基础上进行的，并没有对库函数直接进行测试的必要。

# 有价值的轮子

- 
- 成熟稳定，代码规范，接口友好，功能符合期望；
  - 有相关支持的保证，如文档健全，最好要有实际用例；
  - 社区相对活跃，用户众多，维护频繁，缺陷处理及时；
  - 松耦合，定制容易；
  - 价格合适，有适合的使用许可，最好是开源的。
- 

# 组件

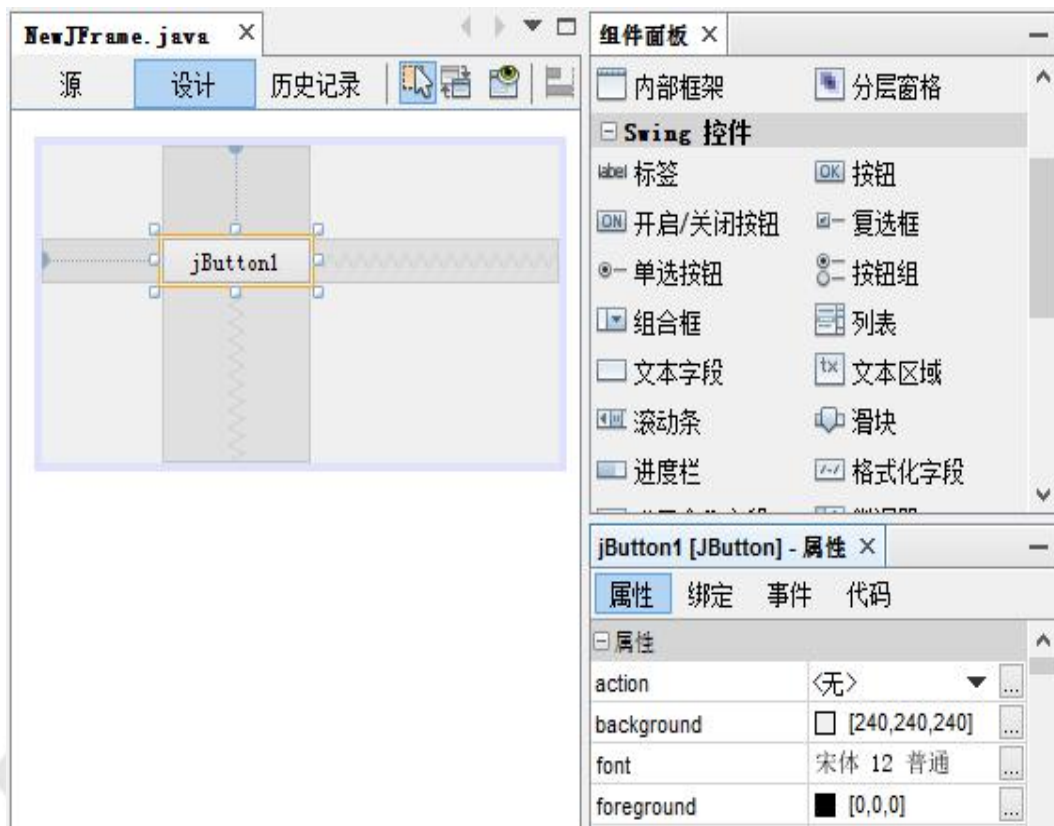
- 组件可以理解作为一种特殊的对象，组件是对数据和方法的简单封装。
- 使用组件可以实现拖放式编程、快速的属性处理以及真正的面向对象的设计。
- 组件是对类库思想的进一步提升，不是仅提供单一类的功能，而是将某个子应用封装提供使用。
- 组件可以对接口进行实现，从而提供实现了这些接口的一类对象。
- 使用现成的组件来开发应用程序时，组件一般可以工作在两种模式下：设计时态和运行时态。

- 在设计时态下，组件显示在窗体编辑器下的一个窗体中。设计时态下组件的方法不能被调用，组件不能与最终用户直接进行交互操作，也不需要实现组件的全部功能。
- 在运行状态下，组件工作在一个已经实际运行的应用程序中。组件必须能够正确地将自身表示出来，它需要对方法的调用进行处理并实现与其他组件之间有效的协同工作。
- 设计时态下所有的组件在窗体中都是可见的，但在运行时态下不一定可见。如Swing中的JTable、JLabel等在运行时态下就可以设置为不可见，但它们均完成了重要的功能。

# 组件的开发

- 自行开发组件与使用组件进行可视化程序开发存在着极大的不同，要求程序员熟知原有的类库结构，精通面向对象程序设计。
- 设计组件是一项艰苦的工作。对于组件的开发者的，组件是纯粹的代码。
- 组件的开发一般不是可视化的开发过程，而是用C++等工具严格编制代码的工作。
- 创建组件的最大意义在于封装重复的工作，其次是可以扩充现有组件的功能。

# 组件的使用



- 组件的使用是一个相对轻松的工作，除了可以使用组件提供的大量功能外，还可以对它们进行定制。
- 组件的定制通常可以通过配置文件的形式进行，常通过一个配置界面，通过交互的方式对组件中需要改动的属性进行指定。



# Java bean组件

- 在Java领域中也存在具体的组件支持机制，即所谓的Java Bean。
- Java Bean是一种特殊的类，在组织上要遵照一定的设计规则，以能够进行配置以及较容易的与所处环境的连接，类似的还有微软的ActiveX等。
- 组件与系统以及组件之间的通讯一般是按照观察者模式的方式进行组织的。
- Java Bean必须存在一个默认的构造函数，即无参数的构造函数。对于每个含有的实例变量，必须存在其简单的get和与其类型相符的set方法。



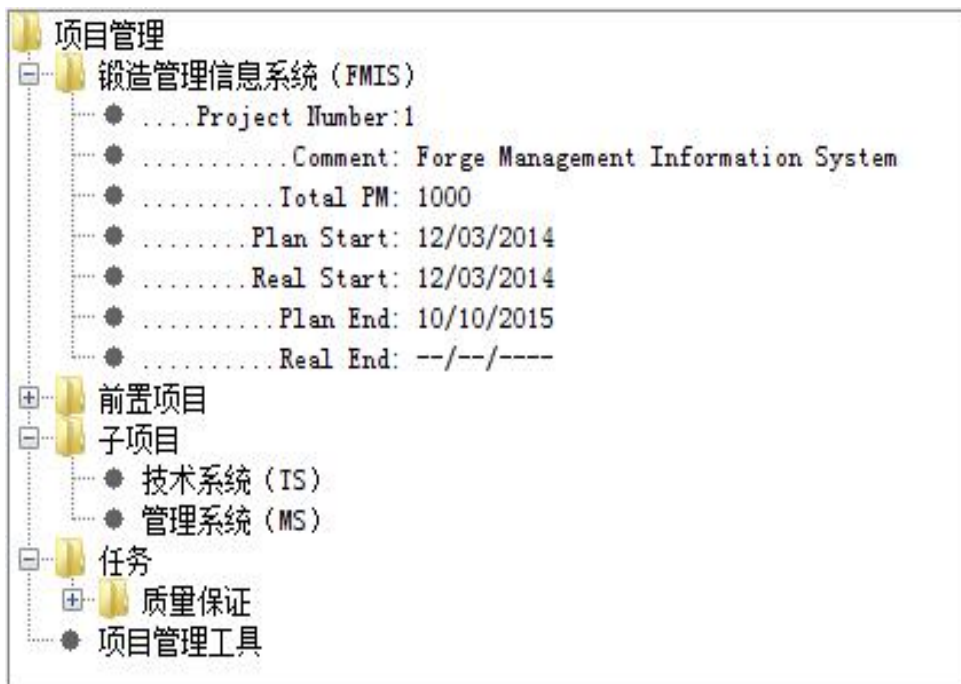
- 该类可实现接口**Serializable**，该接口没有任何函数。一般只能由那些需要对数据进行序列化的类来使用，这样的类通常要求其中的实例变量和类变量（静态变量）的类型也必须实现**Serialization**接口。
- 按照这些要求设计的一个**Java Bean**，如果具有图形化的描述能力，可以直接在**GUI**设计中作为一个可视的组件进行使用。
- 另外，**Java**允许**Java Bean**以一种十分方便的方式对类的对象进行存储和载入。
- **Plain Old Java Object (POJO)**

[对应代码](#)

# 框架

- 快速、高效和正确的将很多原始的工作积累合成到一个更大粒度的半成品式的系统中
- 程序员只需对它进行必要的参数定制就能够将其打造成符合用户需求的真实系统
- 框架提供一个通用平台，通过接口或者类继承的方式嵌入业务类，从而达到系统定制的目的。
- 组件与框架最主要的差别就是控制权在框架中要进行转移，也就是说框架中的类会去调用那些由用户补充实现的对象中的方法，而不会反过来，但这在组件中是会发生的——反射。

# Java中的框架

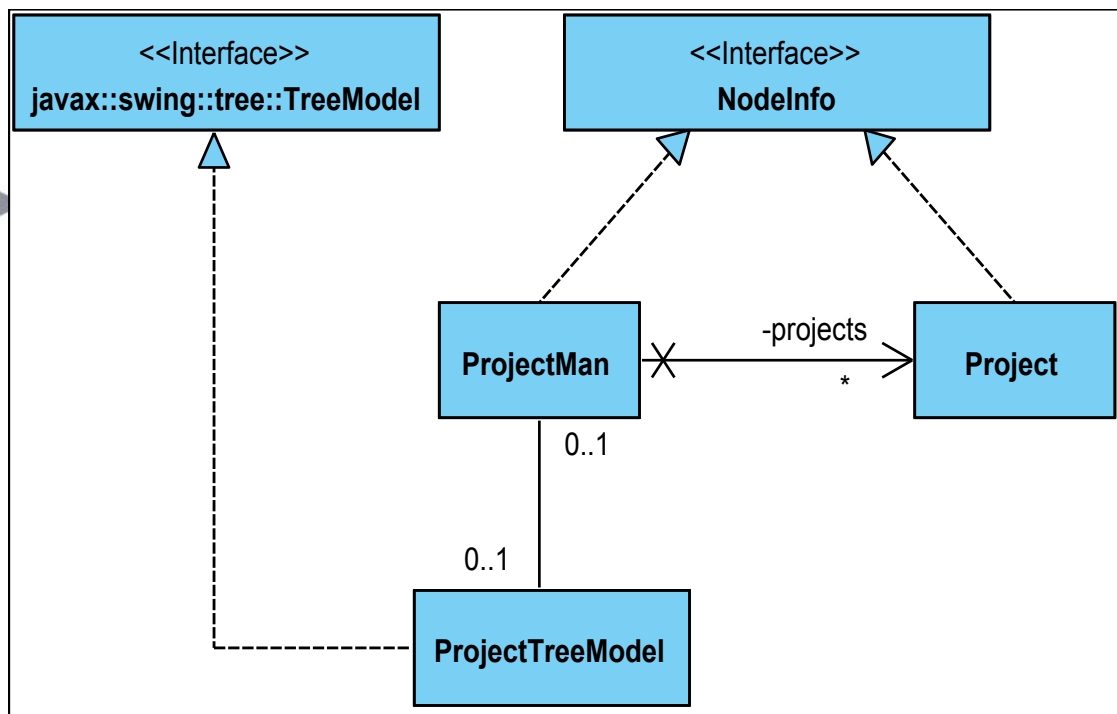


- 考虑一个使用树状结构描述的项目信息
- 图中关于项目、子项目以及前置项目等信息以层次结构进行了展示
- 为了能够使用树的框架，在java中需要一个类对接口 **TreeModel** 进行实现

## Java类TreeModel中的主要方法

<b>void</b>	<b>addTreeModelListener(TreeModelListener)</b> 为树改变后的事件TreeModelEvent添加监听器。
<b>Object</b>	<b>getChild(Object parent, int index)</b> 返回对象parent的孩子数组中索引为index对应的孩子。
<b>int</b>	<b>getChildCount(Object parent)</b> 返回对象parent的孩子数。
<b>int</b>	<b>getIndexOfChild(Object parent, Object child)</b> 返回对象parent中孩子为child的索引值。
<b>Object</b>	<b>getRoot()</b> 返回树的根对象。
<b>boolean</b>	<b>isLeaf(Object node)</b> 返回对象node是否为树的叶子节点。
<b>void</b>	<b>removeTreeModelListener(TreeModelListener l)</b> 删除某个之前使用方法addTreeModelListener加入的监听器。
<b>void</b>	<b>valueForPathChanged(TreePath path, Object newValue)</b> 用户已将path标识的项的值更改为newValue时，进行通知。

## 对应代码



- 类**ProjectTreeModel**，对接口**TreeModel**进行了实现，这个模型类与**ProjectMan**具有关联关系，该类是一个管理类，完成对所有项目的管理。

- 注意到两个类之间的关联是双向导航的，即项目管理类也能够识别类**ProjectTreeModel**，这里实际使用的是一个观察者模式。
- 另外，提供了一个接口**NodeInfo**，是节点信息的抽象描述，项目管理类和项目类分别对其进行了实现。

# 数据的持久化

- 数据的持久化存储一般有两类方式：物理文件或数据库系统。
- 数据库系统的好处在于不同的应用可以在同一时间对同一数据并发的使用，提供了很好的数据共享性和安全性。
- 很多在分布系统中存在的大部分问题都可以通过数据库系统解决。

# 文件持久化

- 直接使用文件的方式进行数据的持久化，不同的程序语言提供了不同的技术支持。
- 其中一个基本功能是能够让开发人员将数据以二进制的原始数据形式将数据存储于文件中。
- 基于这个基本的存储功能，可以衍生出更多的方便的文件存储服务。[对应代码](#)
- 文件的存储方式好处是可读性比较好，可以方便的通过手工编辑或者其它的程序进行读取。
- 问题是开发者必须要为每个需要持久化的类编写存储或读入的方法，这是一项很繁琐的工作。



- 所以一种更直接的想法是，能够将类作为一个整体进行存储，在Java中为了能够进行此项工作，要对进行持久化的类实现接口Serializable，尽管此接口不含有任何方法，只需实现之即可。
- 持久化类要求含有的所有实例变量的类型也都实现了Serializable。[对应代码](#)
- 缺点是存储的文件是二进制形式的，只能由该类所在的应用所识别和读取。
- 而且，当程序中的类发生某些改变时，如增加了一个新的实例变量，再次读取时就会出现问题。
- 为了能够继续从存储的二进制文件中读取数据，必须要通过一个辅助程序将该对象转换为新形式的类型。



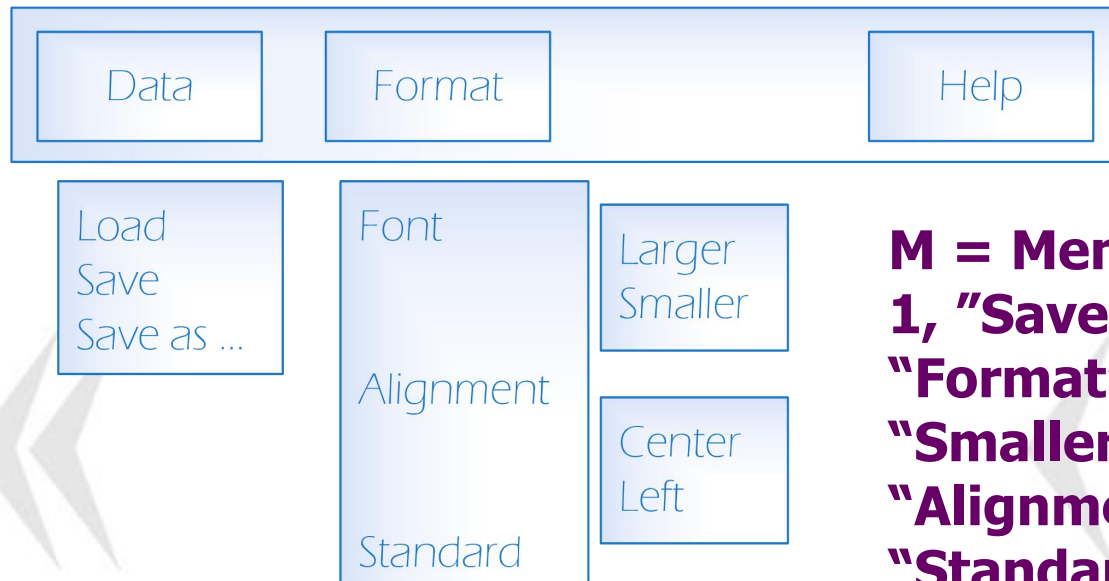
# 数据库持久化

[对应代码](#)

- 大数据量的存储一般情况下都要借助数据库系统来进行。
- 新系统开发大多会选择关系型数据库实现对数据的管理。
- 关系型数据库较为明显的缺点是业务存储的模型需要事先在数据库端设计和构建，在数据持久层需要在应用中进行一些程序设计工作以及对象数据与关系型（表格数据）的兼容性处理等。
- 这种阻抗失配（**Impedence mismatch**）问题的一种解决方式是使用单独的功能软件实现对应用中存在的对象与数据库存储过程的自动化管理，即所谓的对象关系映射（**OR Mapping**）。
- 完成高效管理业务实体类与其持久化形式间的转换，包括事务管理，使开发者专心业务功能开发。

# 领域特定语言

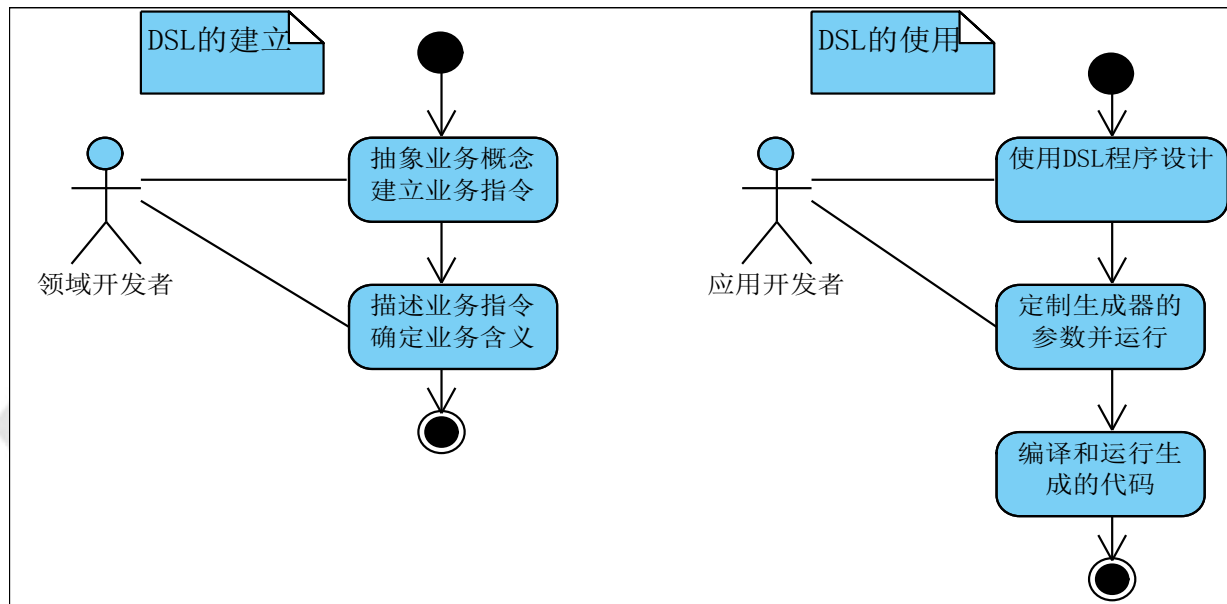
- 领域特定语言（Domain Specific Language, DSL）的初衷是为不同的领域补充进特定的不依赖具体编程语言的抽象指令，或者说针对某一特定领域具有受限表达性的一种计算机程序设计语言。



- 受限性是相对通用编程语言而言

```
M = Menu("Data" Menu("Load" 1, "Save" 2, "Save as..." 3),  
"Format" ("Font" ("Larger" 4, "Smaller" 5),  
"Alignment" ("Center" 6, "Left" 7),  
"Standard" 8), "Help" 9);
```

- DSL的核心价值在于更加清晰地就系统某部分的意图进行表达和沟通。
- DSL提供了一种清晰而准确的概念性的语言，可以有效地改善这种沟通。
- DSL语言另外一个关键的作用是满足平台无关性的要求。
- 重新定义一个全新的语言体系是比较困难的（外部DSL），因此可以借助某种已经熟悉的语言对业务命令进行组织和定义（内部DSL）。

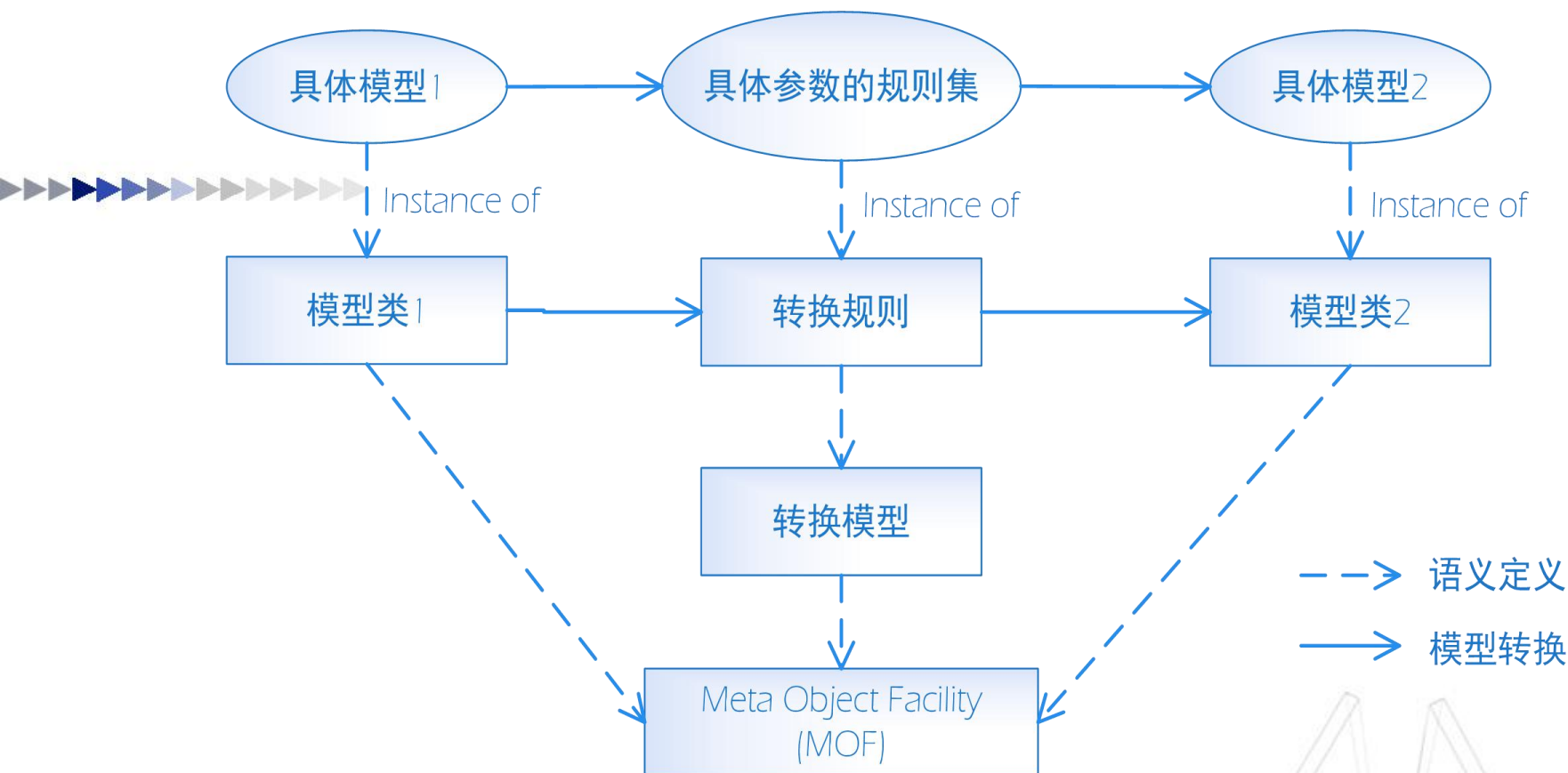


- 外部DSL是一种“不同于应用系统主要使用语言”的语言
- 内部DSL是一种通用语言的特定用法。

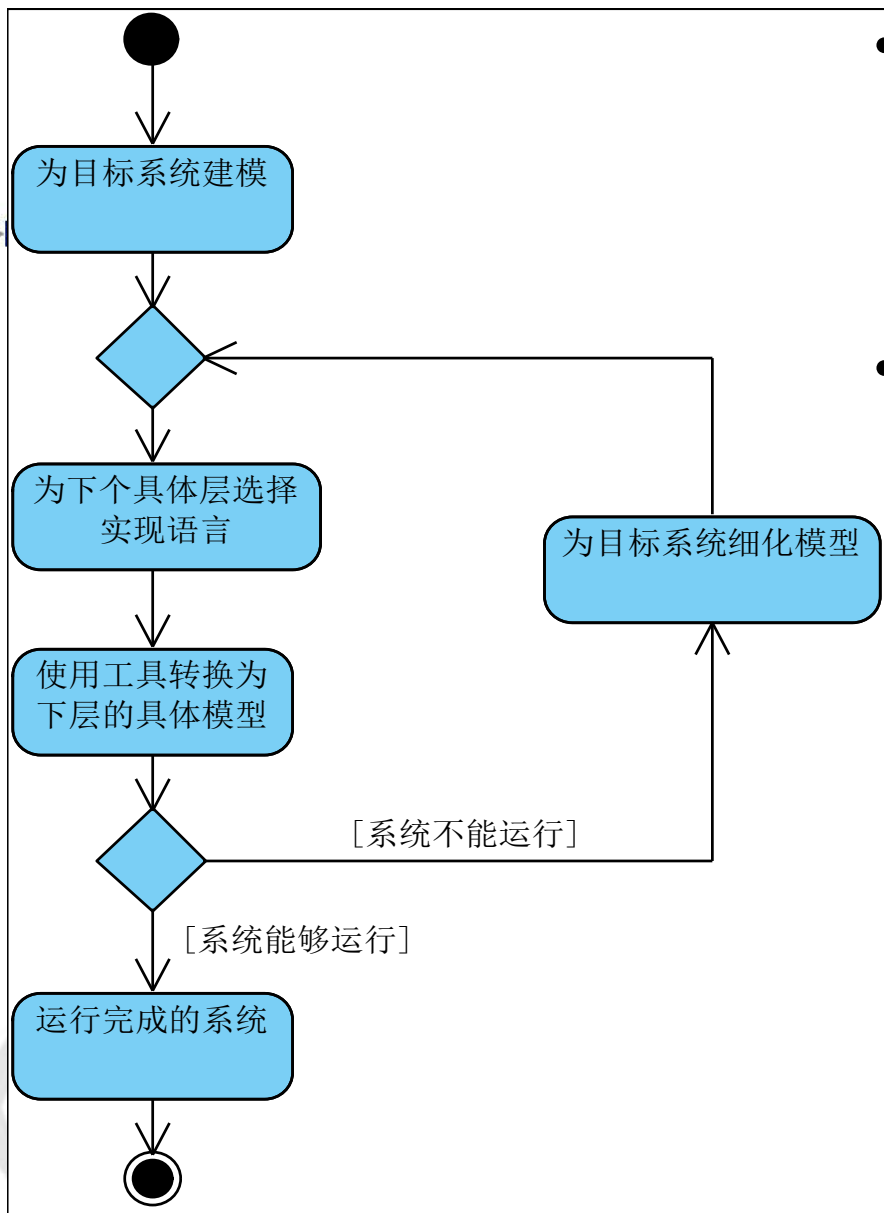
# 模型驱动架构

- 模型驱动架构（MDA）的基本思想是提供一种正式的解决方案，与具体编程语言甚至是架构无关的。
- 使用MDA进行开发的过程，可以分为四个阶段：
  - **CIM（Computation Independent Model）**：聚焦于系统环境及需求，但不涉及系统内部的结构与运作细节。
  - **PIM（Platform Independent Model）**：聚焦于系统内部细节，但不涉及实现系统的具体平台。
  - **PSM（Platform Specific Model）**：聚焦于系统落实于特定具体平台的细节，如EJB，J2EE或.NET都是一种具体平台。
  - **Coding**：最后程序员依据PSM的UML模型内容，按图施工，编写出适用于特定具体平台的代码。

- **MDA**描述的软件开发生命周期和传统生命周期没有大的不同，主要的区别在于开发过程创建的工件，包括**PIM**、**PSM**和代码。
- **PIM**是具有高抽象层次、独立任何实现技术的模型。**PIM**被转换为一个或多个**PSM**。
- **PSM**是为某种特定实现技术量身定做的模型，例如，**EJB PSM**是用**EJB**结构表达的系统模型。
- 开发的最后一步是把每个**PSM**转换为代码，**PSM**同应用技术密切相关。
- **MDA**的变换都是由工具自动完成的，这个过程转换规则的定义也是非常重要的，即要对模型间的过渡进行形式化的描述。



- 图中将PIM和PSM的转换描述为模型1到模型2的过渡。
- 所有模型和转换规则需要正式的语义定义，因此OMG组织出台了一系列的规范将MDA方法标准化，其中最重要的语义描述包含在规范Meta Object Facility (MOF) 中。

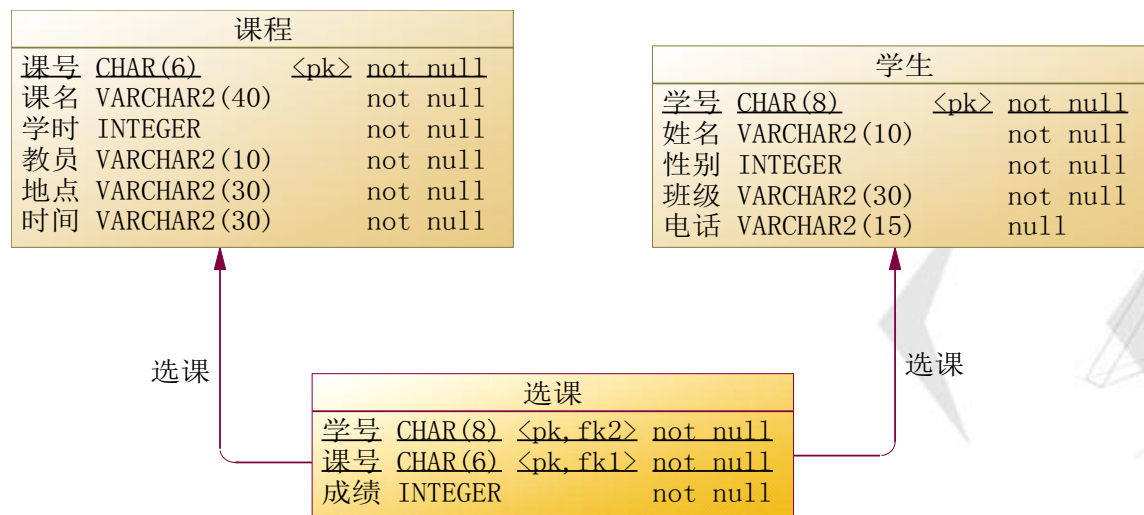
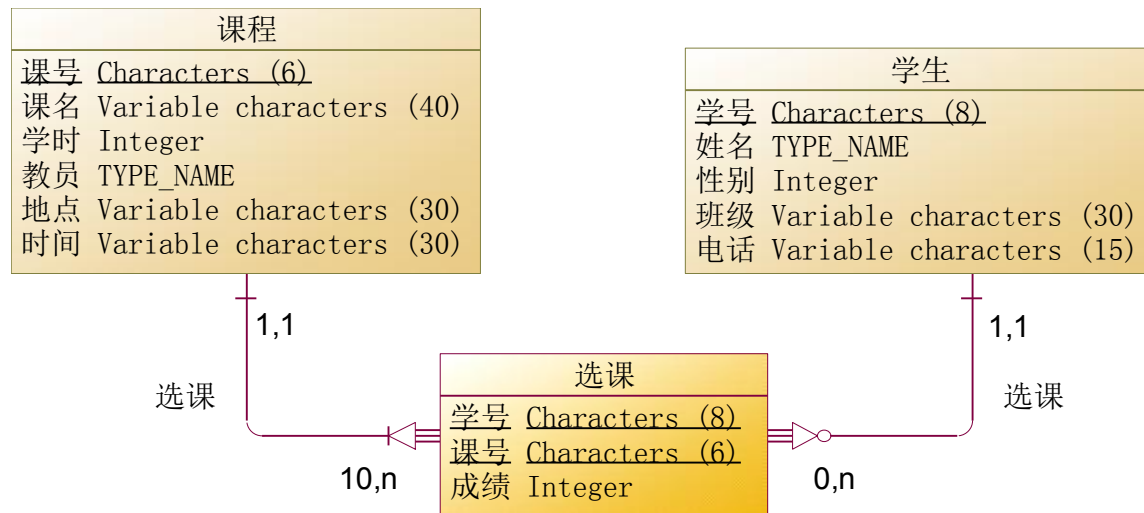
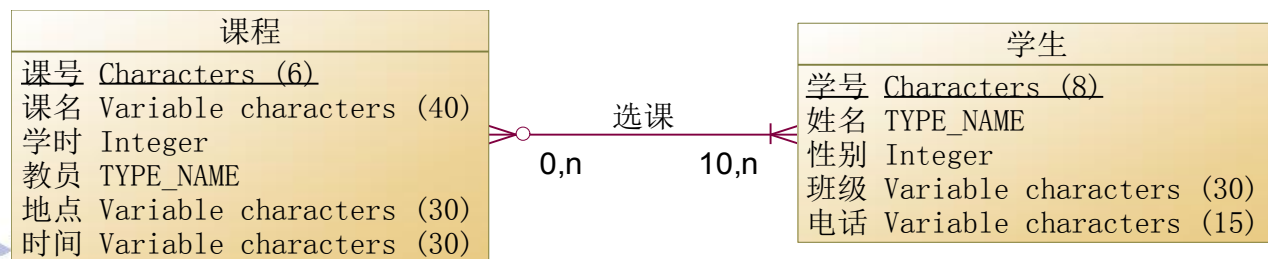


- MDA的一个具体的应用是在数据库设计工具Power Designer中针对实体关系图(ERD)的构建。

- 数据库模型的设计依次分为四个主要的阶段，分别为概念模型（Conceptual Design Model, CDM）、逻辑模型（Logical Design Model, LDM）、物理模型（Physical Design Model, PDM）和数据库代码，分别对应为MDA中的CIM、PIM、PSM和Coding四个层次。



## 对应代码



- 模型CDM和LDM是与具体数据库平台无关的模型，
- 其中含有的字段类型也是中立类型，并由设计工具负责维护到具体数据库平台的映射规则。
- PDM是在Oracle 10g平台上对应的物理模型




# 重构

- 复杂程序的可重构性是面向对象的目标之一，为了保证此目标的实现，必须要尽可能的对程序进行简化以增加其可读性。
- 如在详细设计文档中除了对方法在行为上的要求外，在方法的具体编码上还可约定额外的准则：
- 方法的名字要尽可能的自解释；
- 方法最长应不超过12行，尽可能少的包含while、switch和if逻辑块。

- 如何对复杂的方法进行简单化？
- 为此，代码重构发挥了作用，它给出了一些系统化的针对方法级别的简化方法。
- 在方法的级别上主要考虑的就是如何将一个语法正确的程序片段整合为一个新的方法。

```
public int ref ( int x, int y, int z) {  
    int a=0;  
    if (x>0) {  
        a=x;  
        x++;  
        --y;  
        a = a + y + z;  
    }  
    return a;  
}
```

```
public int ref ( int x, int y, int z) {  
    int a=0;  
    if(x>0){  
        a = doit(x, y, z);  
    }  
    return a;  
}  
private int doit (int x, int y, int z) {  
    int a;  
    a=x;  
    x++;  
    --y;  
    a=a+y+z;  
    return a;  
}
```



```
public int ref2 (int x){  
    int a=0;  
    int b=0;  
    int c=0;  
    if(x>0){  
        a = x;  
        b = x;  
        c = x;  
    }  
    return a+b+c;  
}
```

```
int Computation::ref2 (int x) {  
    int a=0;  
    int b=0;  
    int c=0;  
    if(x>0){  
        abcModify(a,b,c,x);  
    }  
    return a+b+c;  
}  
  
void Computation::abcModify(int& a,  
int& b, int& c, int x){  
    a=x;  
    b=x;  
    c=x;  
}
```

- 重构允许在程序首次构造后对其结构进行改造，使其能够按照希望的形式进行构造。
- 重构后代码的可读性和可重用性更强，其思想可同样扩展到对类的再造（reconstruction）和改造（outsourcing）上。

# 作业

- 习题1、2、3

