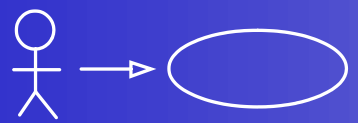
软件需求设计UML全程实作

设计

潘加宇



建模工作流

*业务建模
- 愿景
- 选定组织
- 业务用例图
- 现状业务序列图
- 改进业务序列图

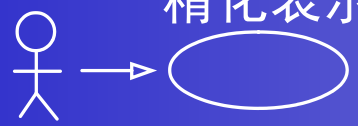
*需求
- 系统用例图
- 系统用例规约

*分析
- 分析类图
- 分析序列图
- 分析状态机图

*设计
- 建立数据层
- 精化业务层
- 精化表示层

需求 提升销售

设计 降低成本

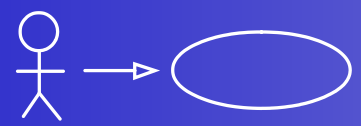


实现

- 可执行模型
- 模型生成代码
- 模型手工编写代码

目前可行：核心域模型＋典型用例代码案例

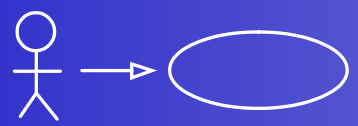
驱动和自动



实现

- 领域逻辑是否集中描述？
- 实体的状态机是否得到维护？

没有固定形式：根据现实折衷

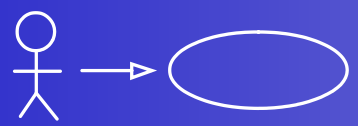


实现

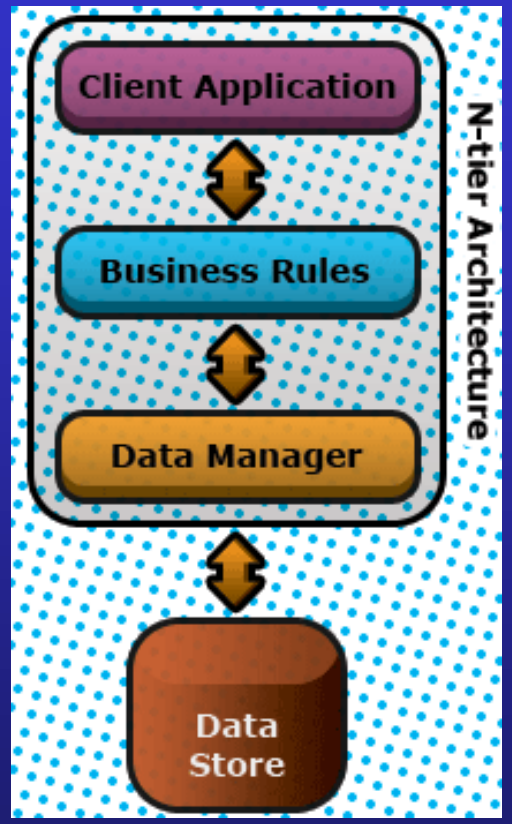
必须提供的服务

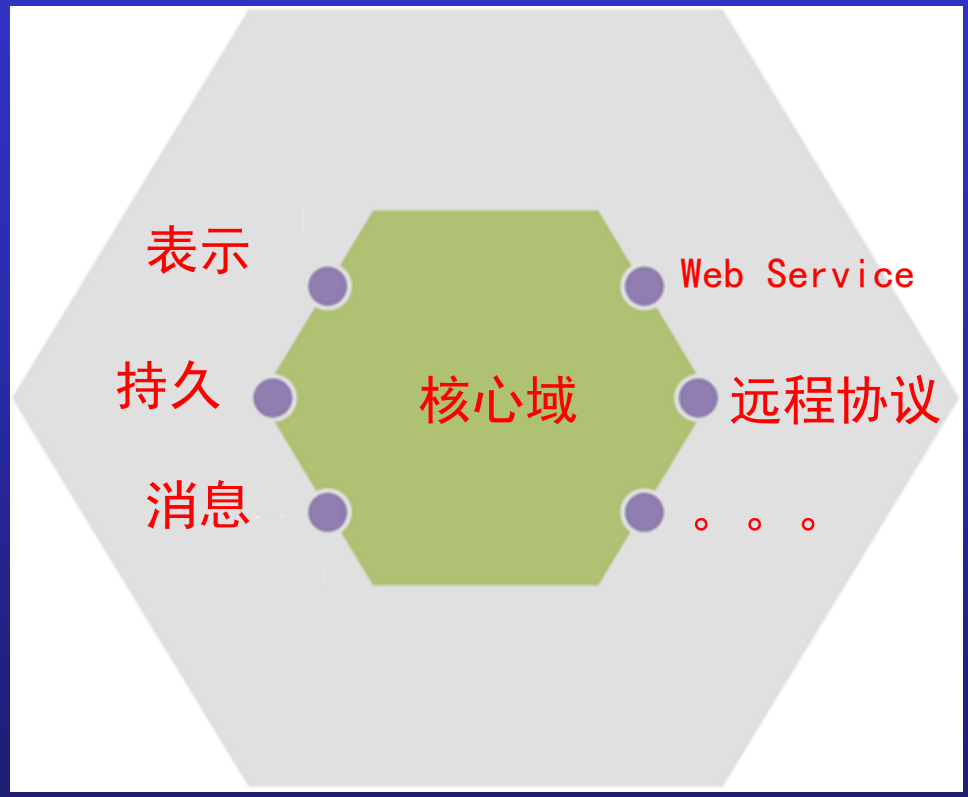
控制（服务）类
UC的映射

界面驱动…数据驱动…



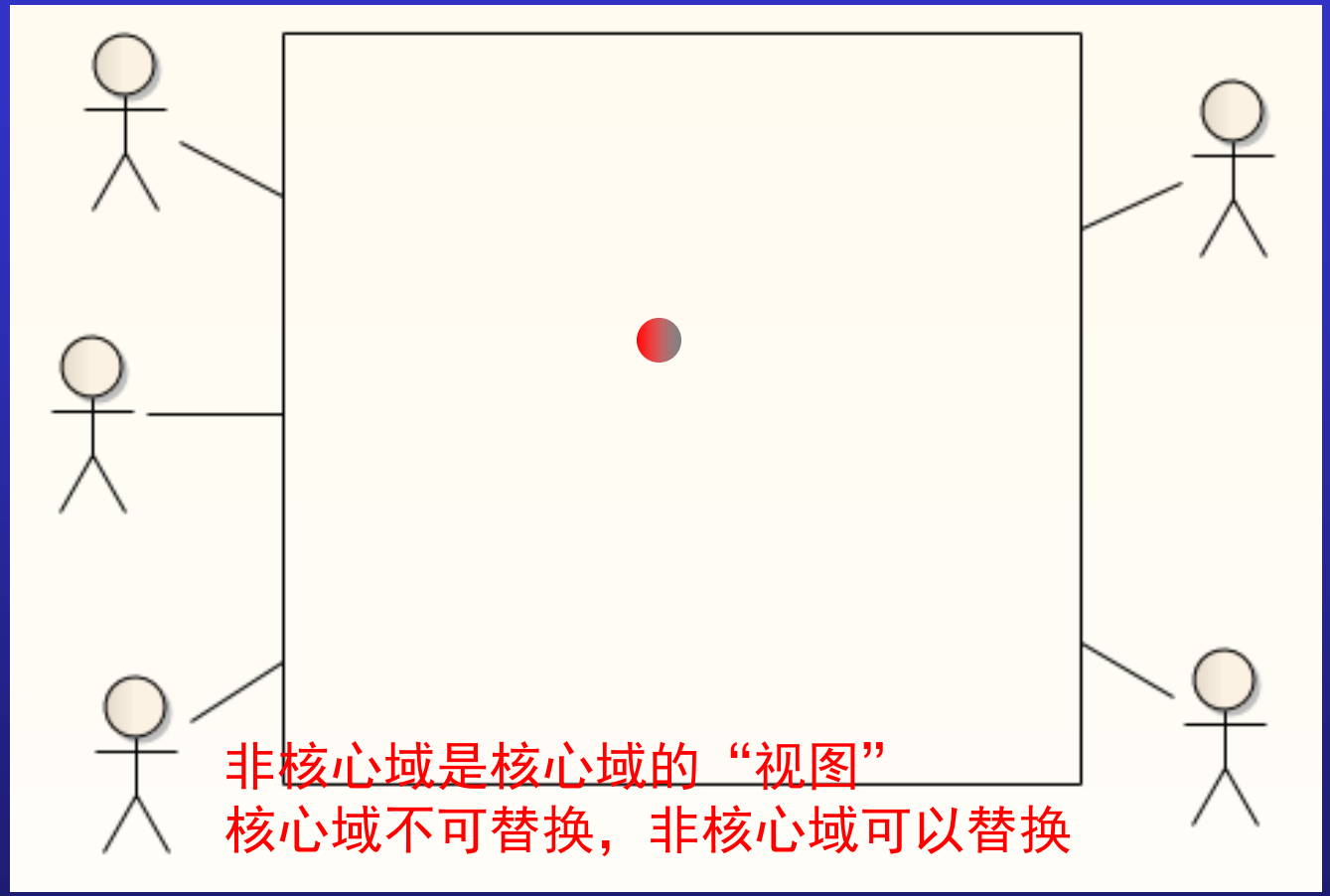
实现





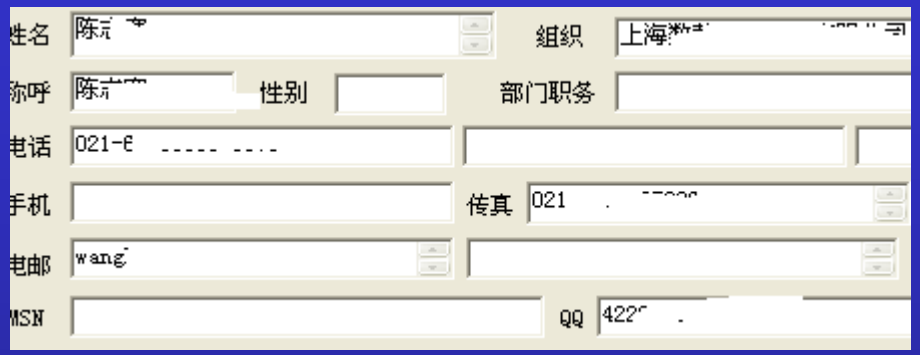
从分层到核心域为中心

实现

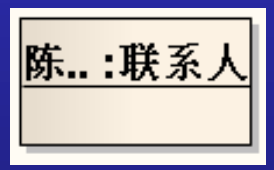


模型和视图

实现

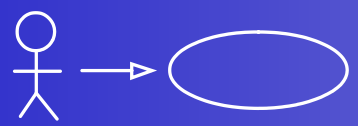


多一个视图

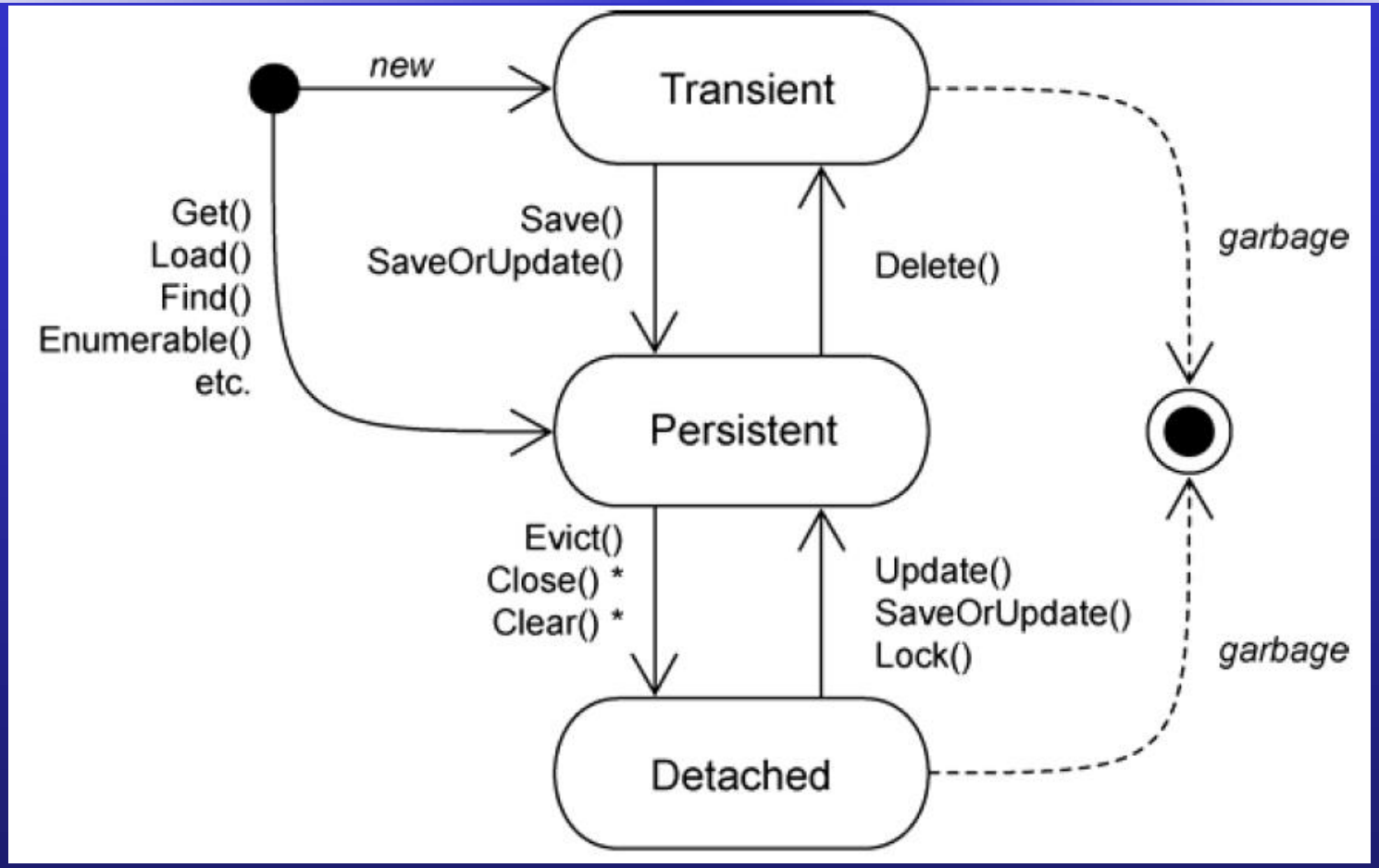


多一个维度的状态机

模型和视图

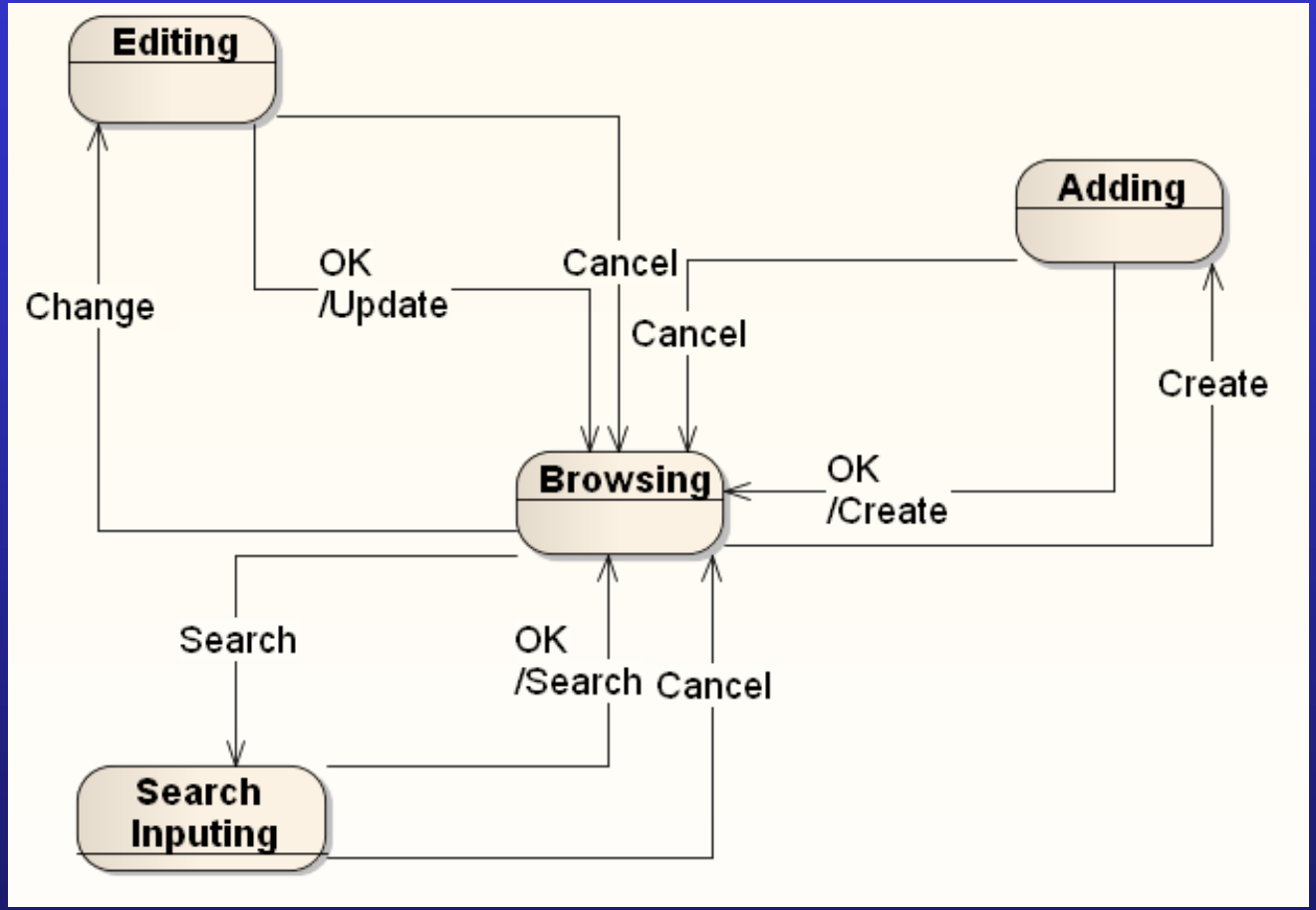


实现



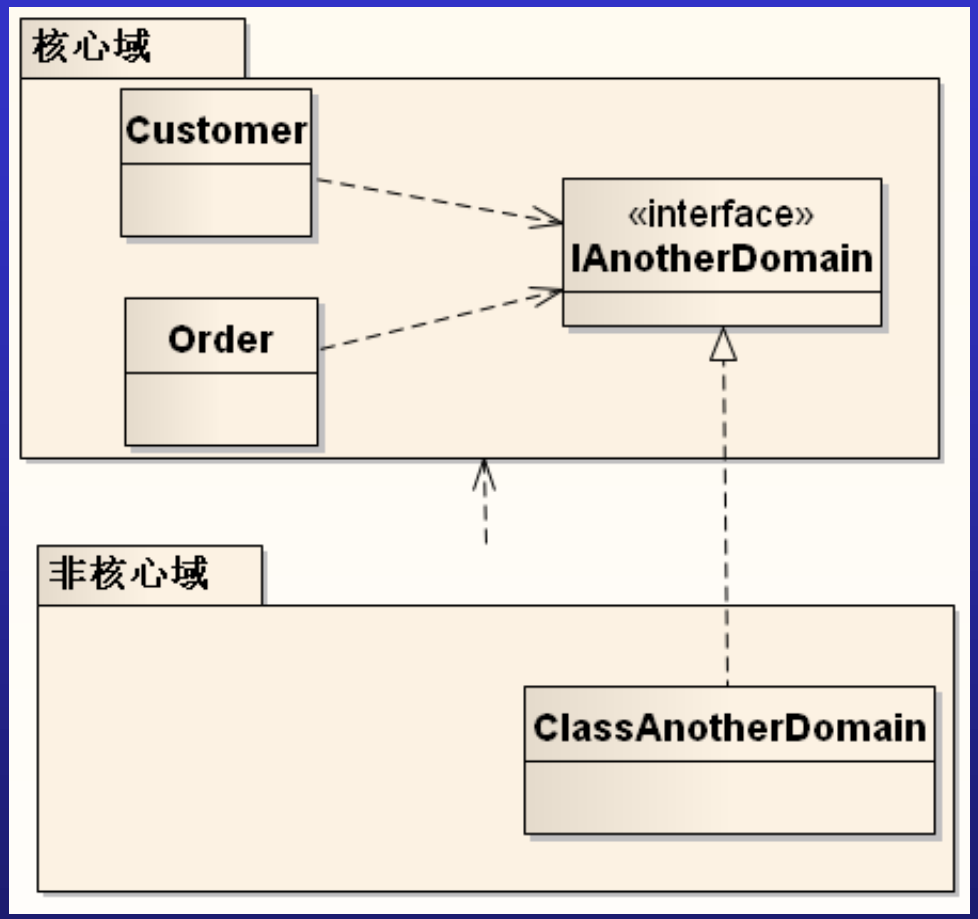
持久维度下的状态机

实现



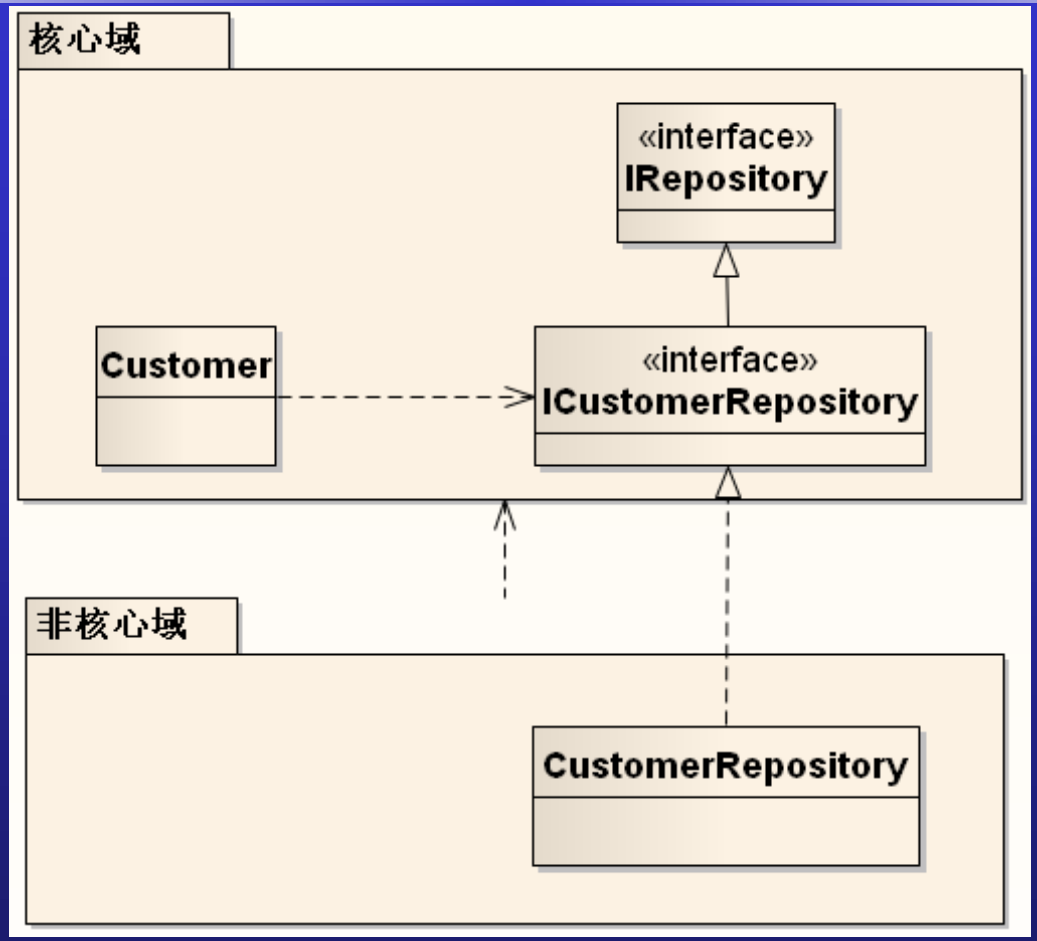
表示维度下的状态机

实现


核心域
Customer
«interface»
IAnotherDomain
Order
非核心域
ClassAnotherDomain

分离接口

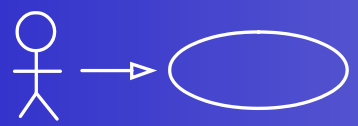
实现


核心域
«interface»
IRepository
Customer
«interface»
ICustomerRepository
非核心域
CustomerRepository

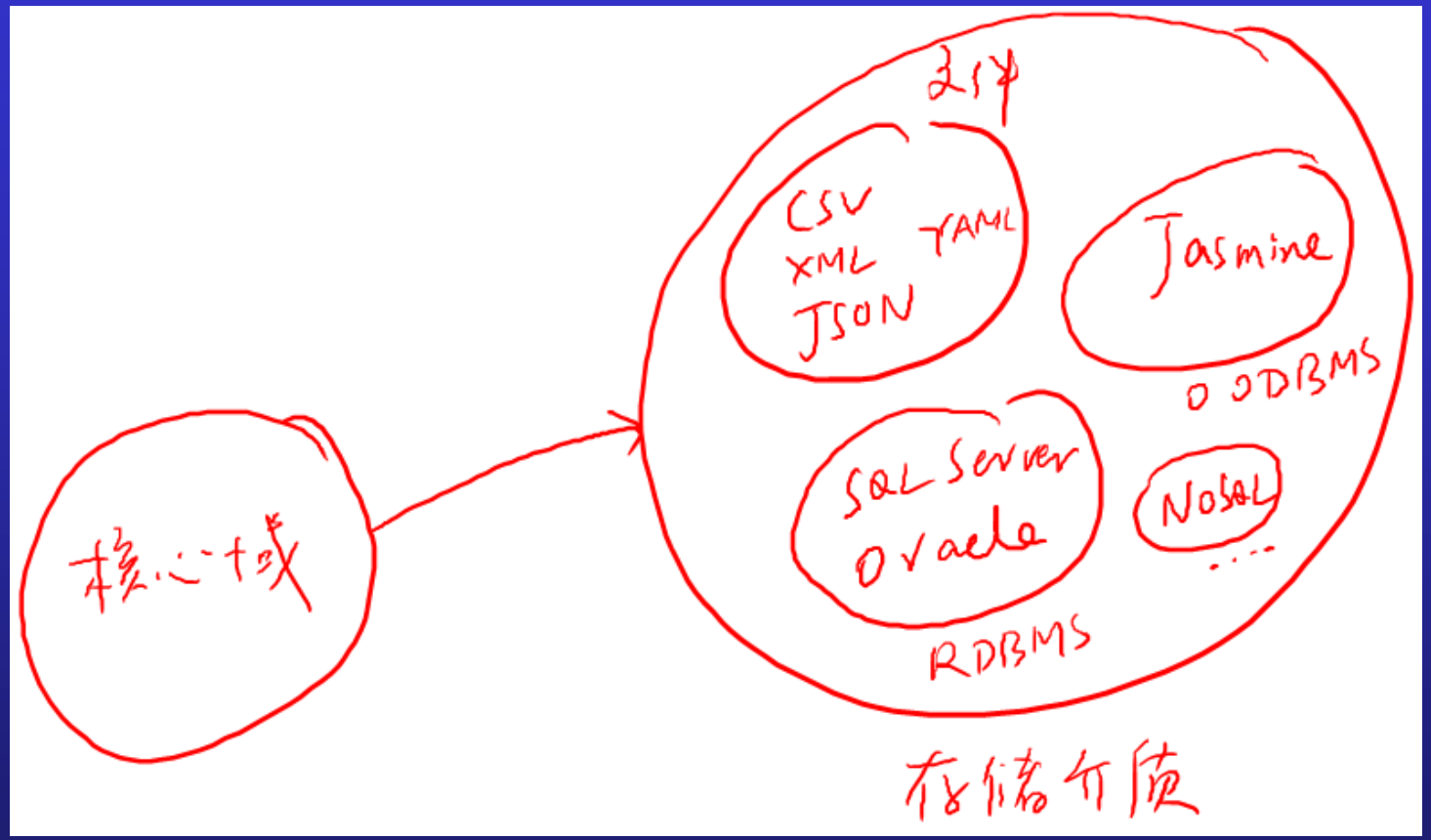
分离接口——持久仓储

数据层

- 映射存储
- 构造数据源层



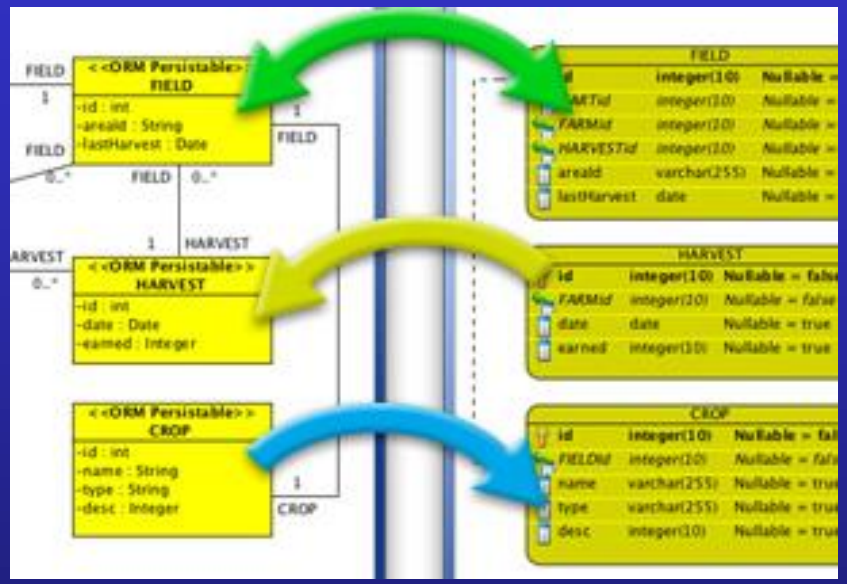
存储


核心域
文件
CSV
XML
JSON
YAML
Jasmine
ODBMS
SQL Server
Oracle
NoSQL
RDBMS
存储介质

核心域映射到存储

存储

对象建模正确
映射出来自然符合范式



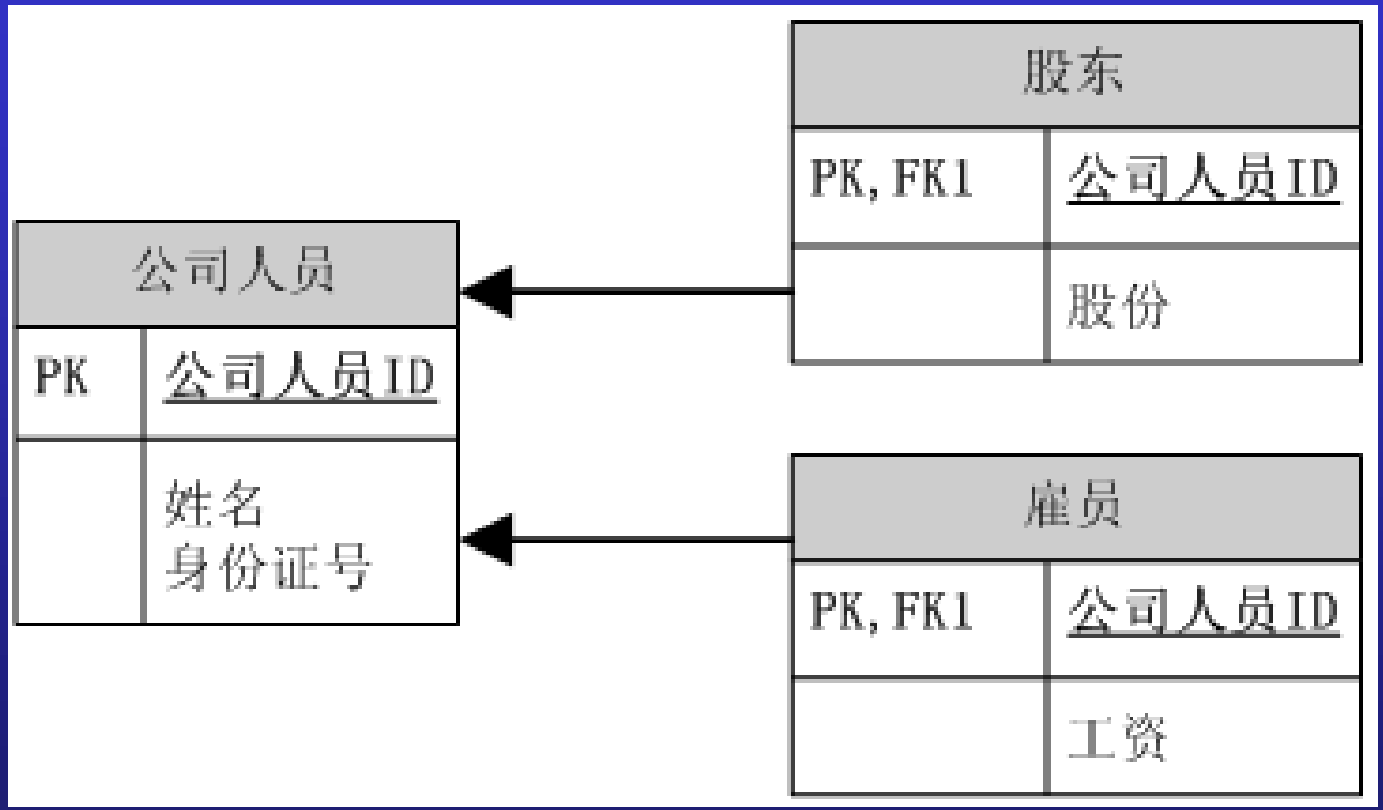
类 —— 表

对象 —— 行

属性 —— 列

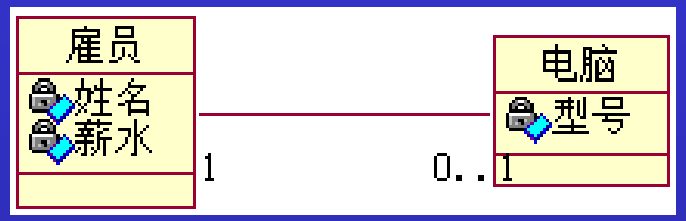
结构映射

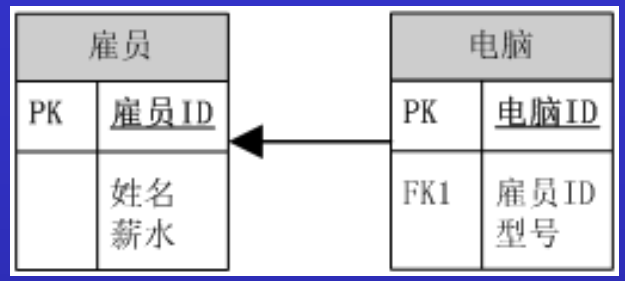
存储

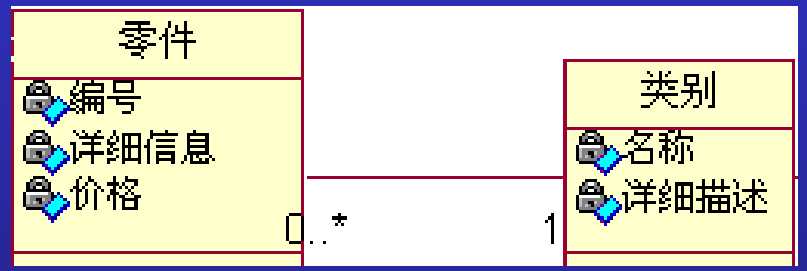


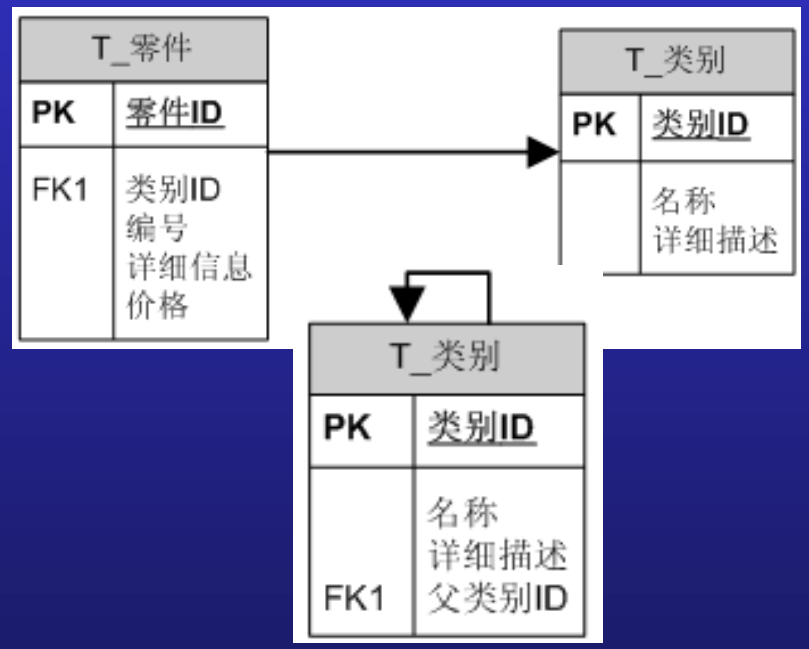
泛化——超类主键作为所有类的主键

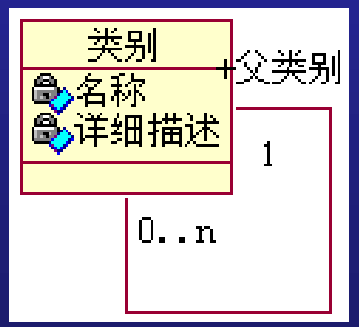
存储



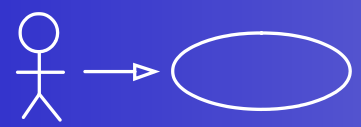




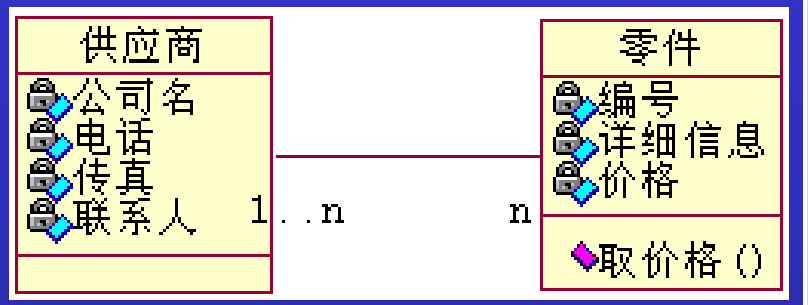


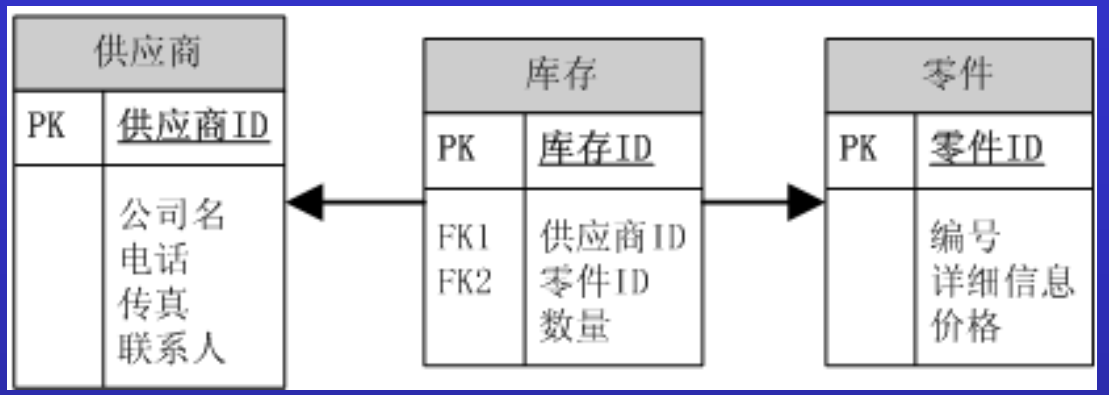


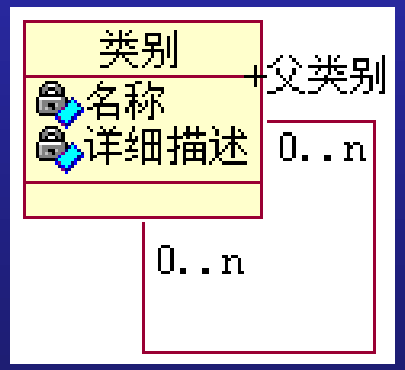
关联——外键放在“非1”端

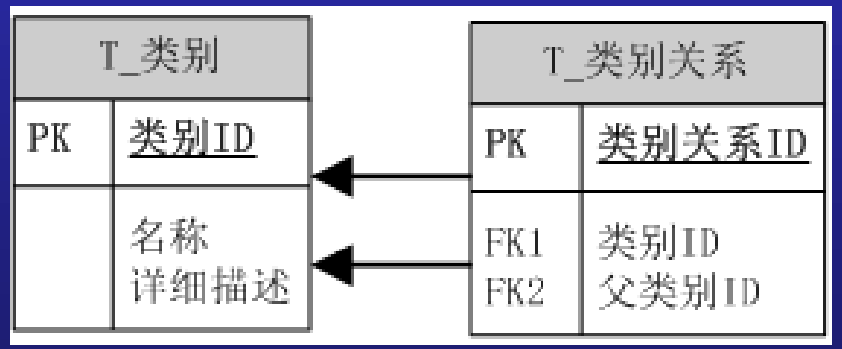


存储

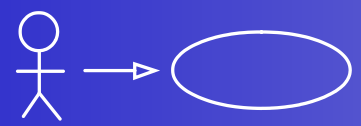




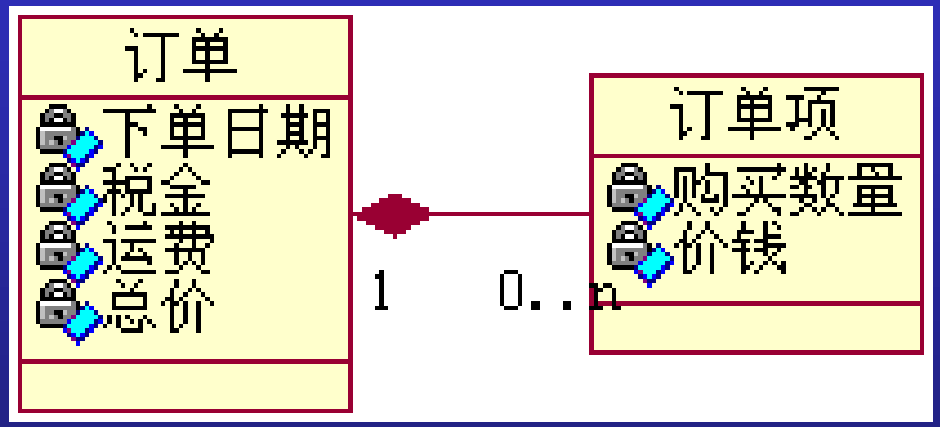


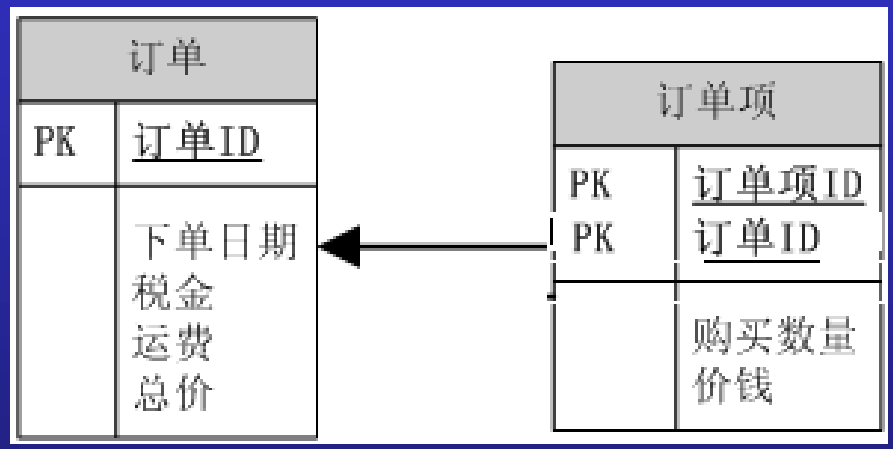


多对多关联——添加关联表



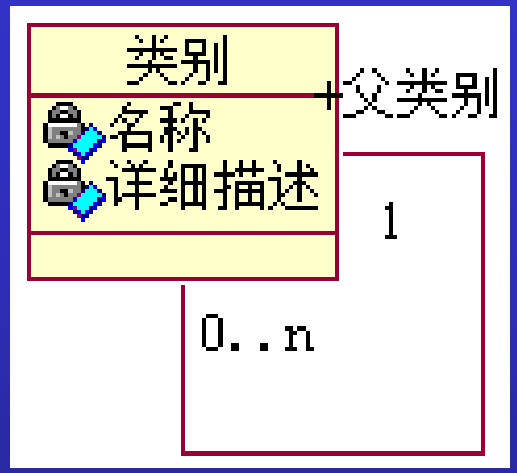
存储

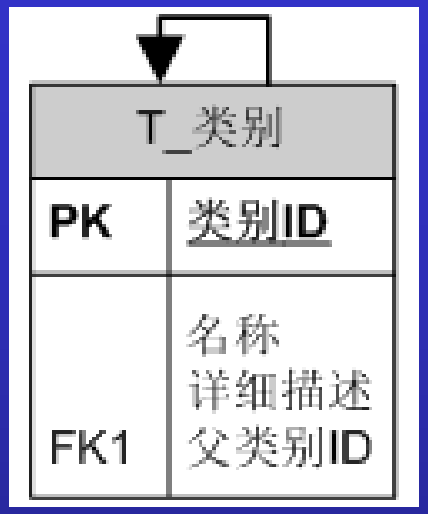

订单
下单日期
税金
运费
总价
1
0..n
订单项
购买数量
价钱


订单
PK
订单ID
下单日期
税金
运费
总价
订单项
PK
订单项ID
PK
订单ID
购买数量
价钱

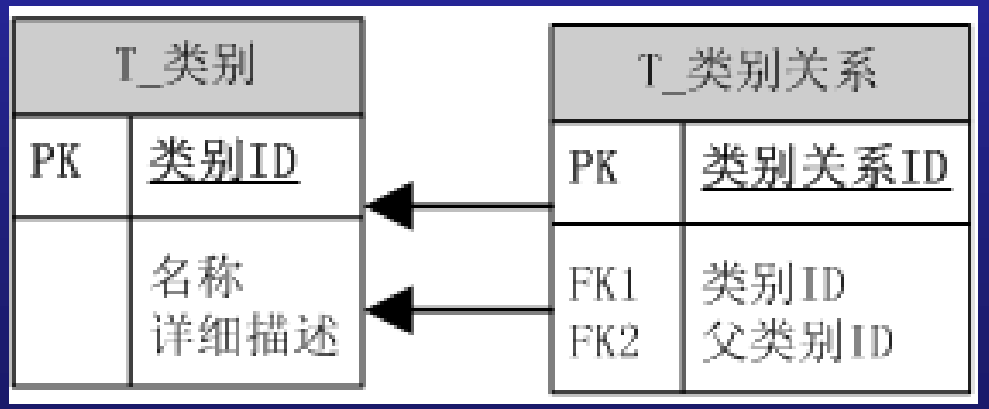
组合

存储


类别
名称
详细描述
+父类别
1
0..n

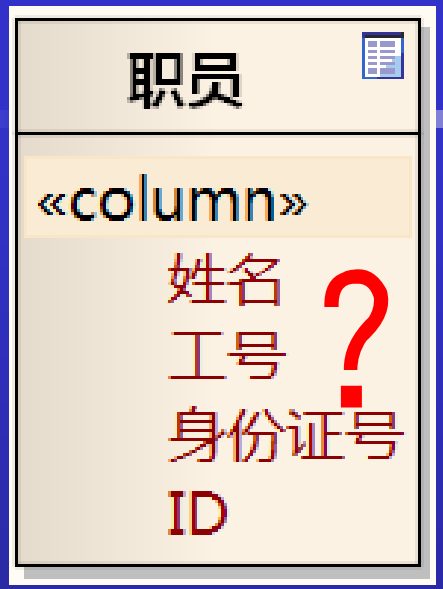

T_类别
PK
类别ID
名称
详细描述
FK1
父类别ID


类别
名称
详细描述
+父类别
0..n
0..n


T_类别
PK
类别ID
名称
详细描述
T_类别关系
PK
类别关系ID
FK1
FK2
类别ID
父类别ID

存储

- 唯一标识记录
- 被其他表引用为外键



- 要求：唯一、恒定

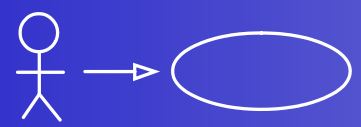
- 有领域含义，意味着可能潜伏着变化
- 把领域字段和主键分开

主键

存储

- 原则
  - 延展性越来越重要
  - 先追求局面上的大赢
  - 出现不可调和的性能问题时，再作微调
  - 大赢使微调更放心

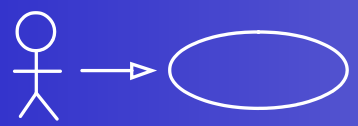
- 非规范化（微调性能，慎用）
  - 冗余列——方便检索（Total，uppercase…）
  - 冗余表（远端外键）



存储

- 更稳定的设计
- 每个表的主键都是相同的数据类型
- 表间连接被限定在单个列上，SQL语句的书写不复杂

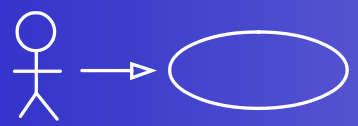
代理主键



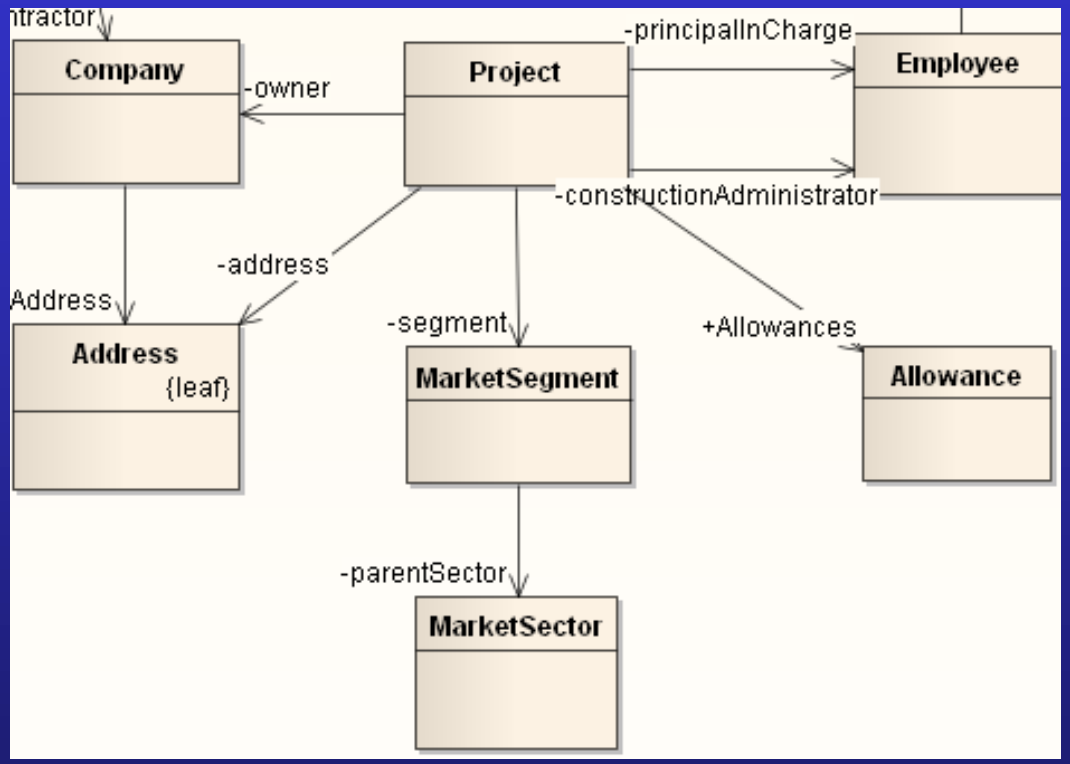
存储

- 隐藏代理主键
  - 不要让用户在屏幕或报表上看见
  - 不要让用户输入

代理主键

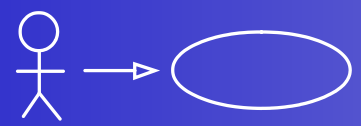


存储

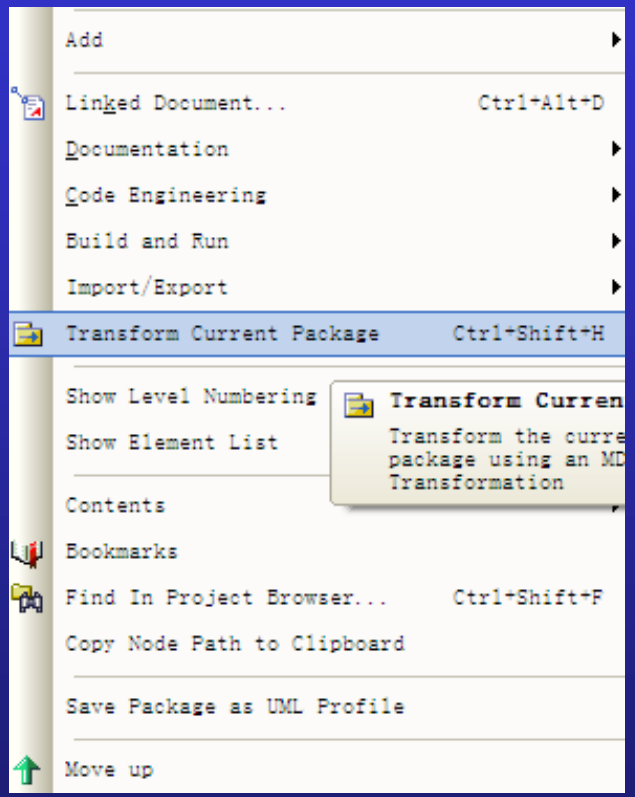


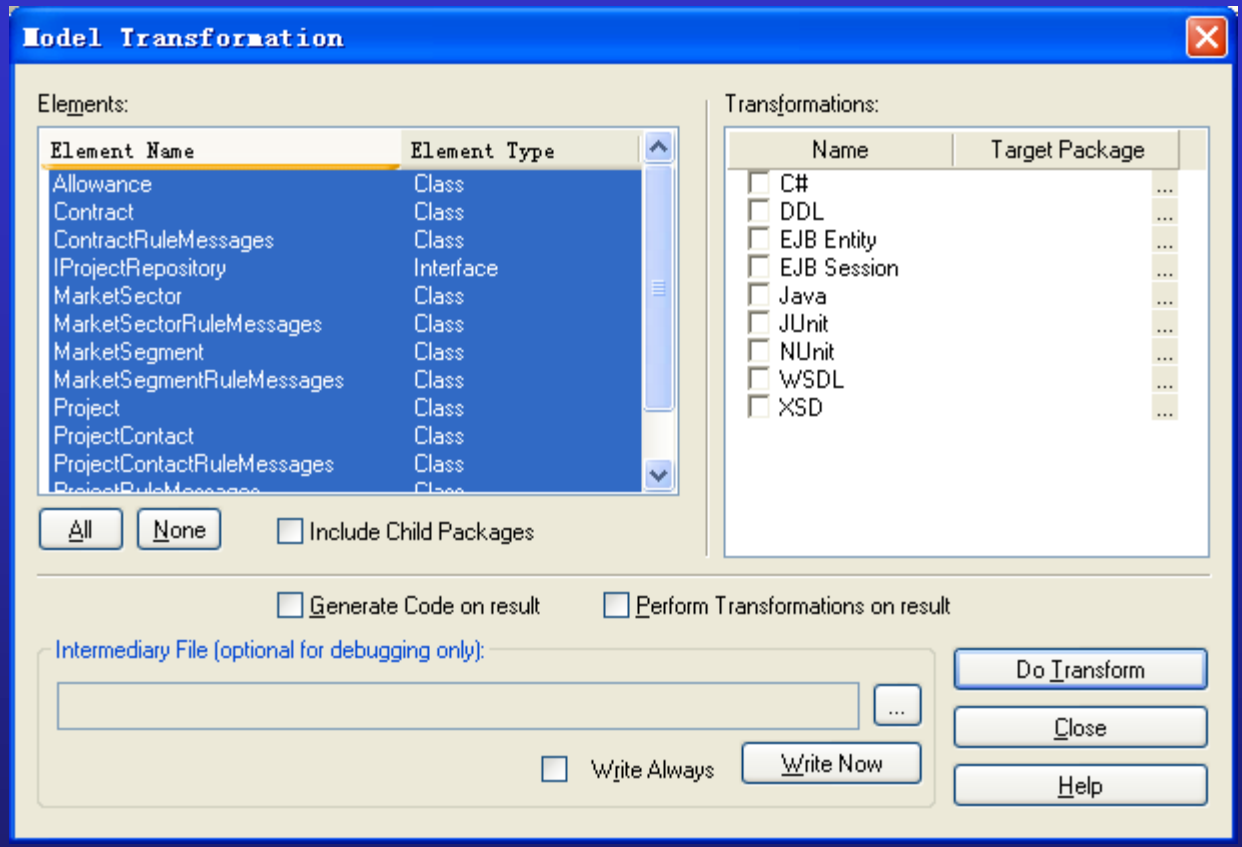

```
public class Project : EntityBase, IAggregateRoot
{
    #region Private Fields

    private string number;
    private string name;
    private Address address;
    private Company owner;
    private MarketSegment segment;
    private List<Allowance> allowances;
    private List<Contract> contracts;
    private List<ProjectContact> contacts;
    private Employee constructionAdministrator;
    private Employee principalInCharge;
```

实体类图



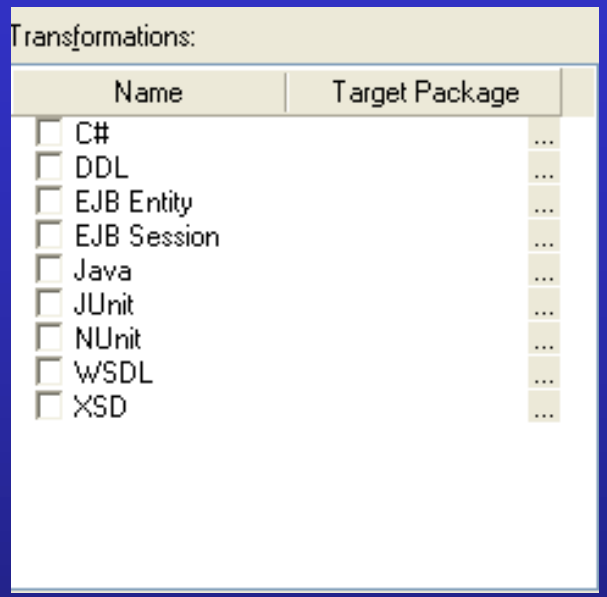
存储

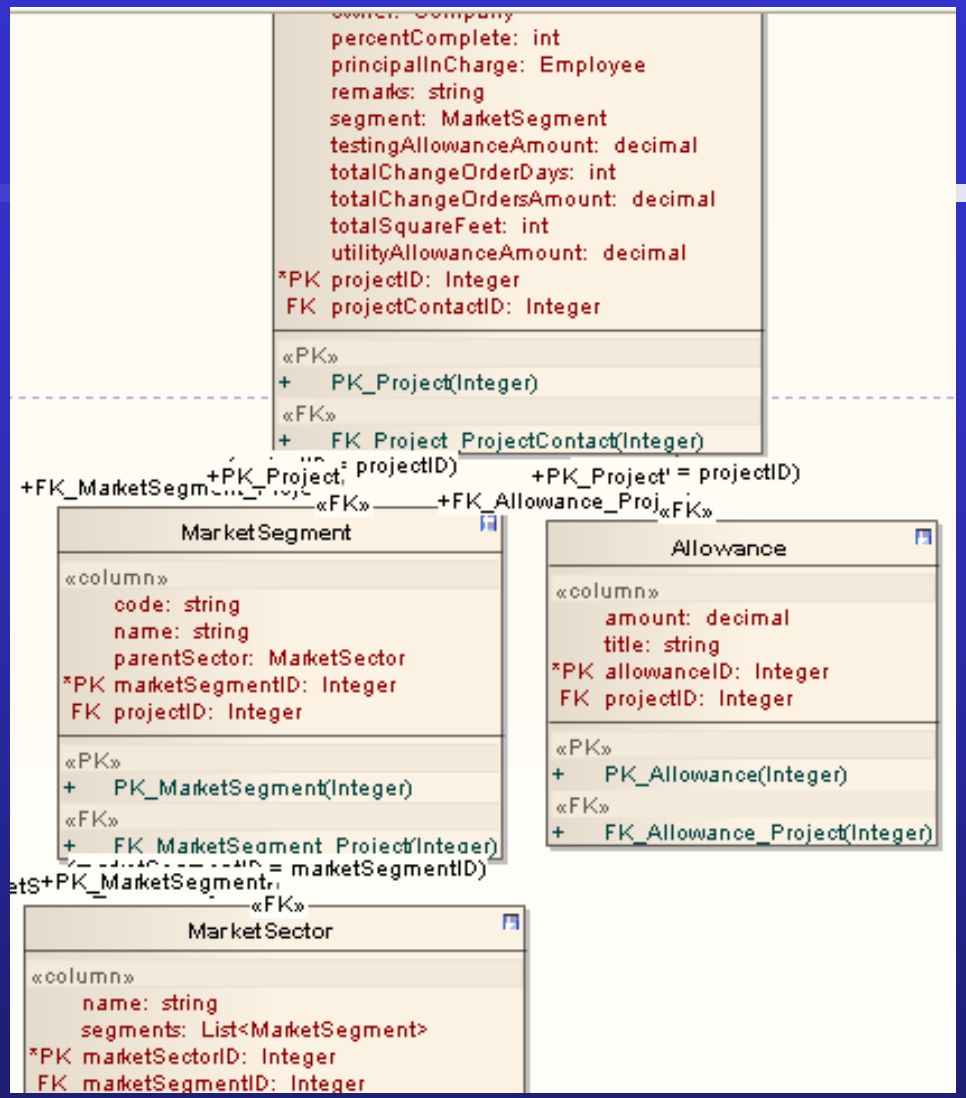

Add
Linked Document... Ctrl+Alt+D
Documentation
Code Engineering
Build and Run
Import/Export
Transform Current Package Ctrl+Shift+H
Show Level Numbering
Show Element List
Transform Curren
Transform the curre
package using an MD
Transformation
Contents
Bookmarks
Find In Project Browser... Ctrl+Shift+F
Copy Node Path to Clipboard
Save Package as UML Profile
Move up


Model Transformation
Elements:
Element Name
Element Type
Allowance Class
Contract Class
ContractRuleMessages Class
IProjectRepository Interface
MarketSector Class
MarketSectorRuleMessages Class
MarketSegment Class
MarketSegmentRuleMessages Class
Project Class
ProjectContact Class
ProjectContactRuleMessages Class
All
None
Include Child Packages
Transformations:
Name
Target Package
C#
DDL
EJB Entity
EJB Session
Java
JUnit
NUnit
WSDL
XSD
Generate Code on result
Perform Transformations on result
Intermediary File (optional for debugging only):
Write Always
Write Now
Do Transform
Close
Help

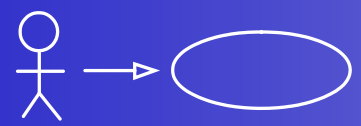
转换

存储


Transformations:
Name
Target Package
C#
DDL
EJB Entity
EJB Session
Java
JUnit
NUnit
WSDL
XSD

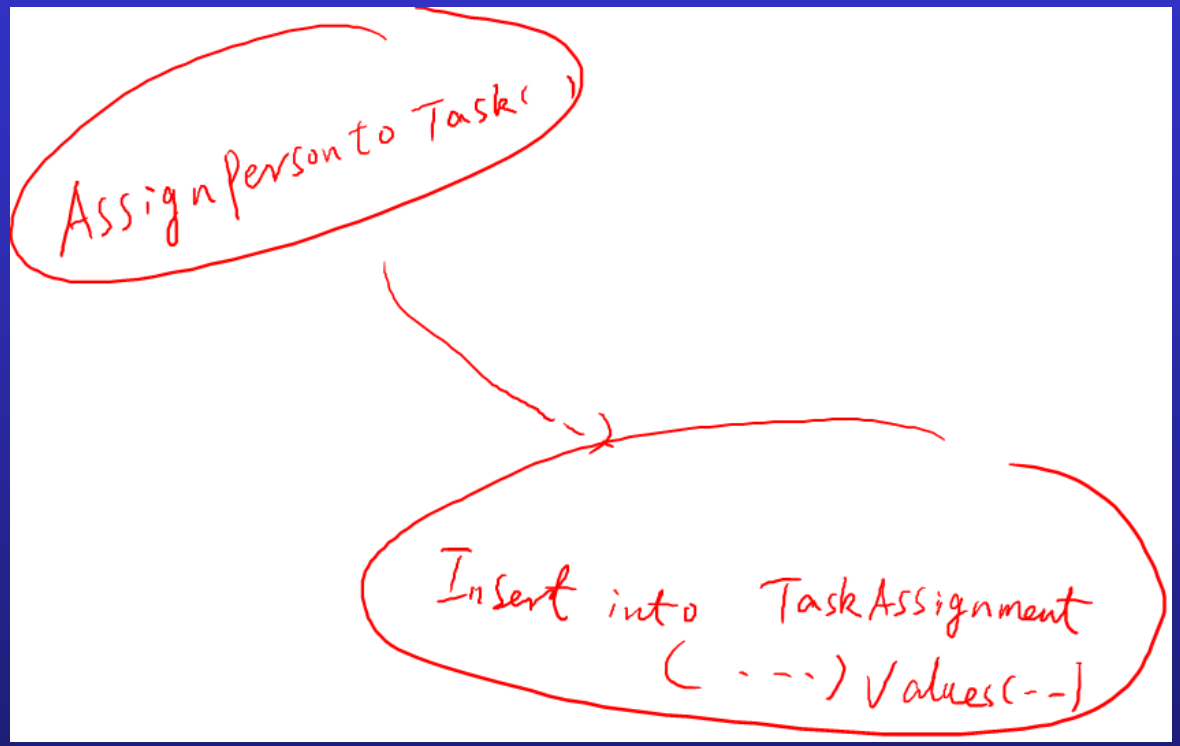

percentComplete: int
principalInCharge: Employee
remarks: string
segment: MarketSegment
testingAllowanceAmount: decimal
totalChangeOrderDays: int
totalChangeOrdersAmount: decimal
totalSquareFeet: int
utilityAllowanceAmount: decimal
*PK projectID: Integer
FK projectContactID: Integer
«PK»
+ PK_Project(Integer)
«FK»
+ FK_Project_ProjectContact(Integer)
+PK_Project = projectID)
+PK_Project = projectID)
«FK»
«FK»
MarketSegment
«column»
code: string
name: string
parentSector: MarketSector
*PK marketSegmentID: Integer
FK projectID: Integer
«PK»
+ PK_MarketSegment(Integer)
«FK»
+ FK_MarketSegment_Project(Integer)
Allowance
«column»
amount: decimal
title: string
*PK allowanceID: Integer
FK projectID: Integer
«PK»
+ PK_Allowance(Integer)
«FK»
+ FK_Allowance_Project(Integer)
+PK_MarketSegment = marketSegmentID)
«FK»
MarketSector
«column»
name: string
segments: List<MarketSegment>
*PK marketSectorID: Integer
FK marketSegmentID: Integer

映射DDL

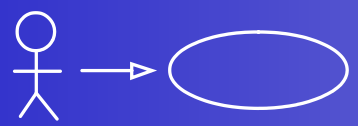


数据源层

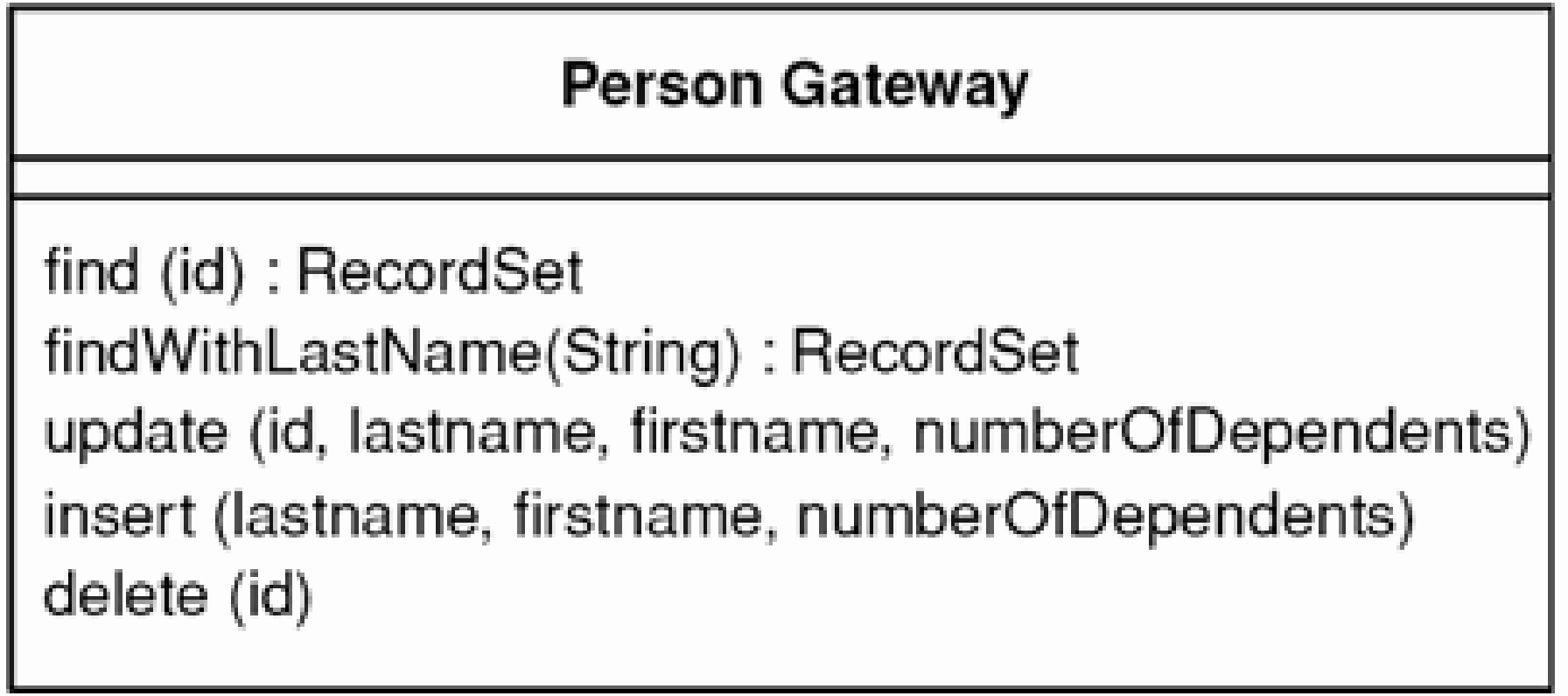
- 表数据入口
- 行数据入口
- 活动记录
- 数据映射器



领域逻辑和存储逻辑的转换



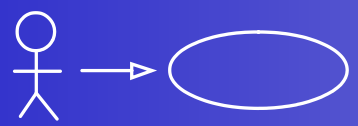
表数据入口



一个实例处理表中所有行

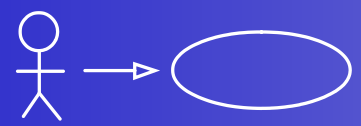
表数据入口

- 把所有和一张表有关的SQL放在同一个类
- 这个类只有一个对象
- 适用于表模块、事务脚本
- 操作
  - findAll, findPerson, findWithAge, ...
  - insert, delete, ...

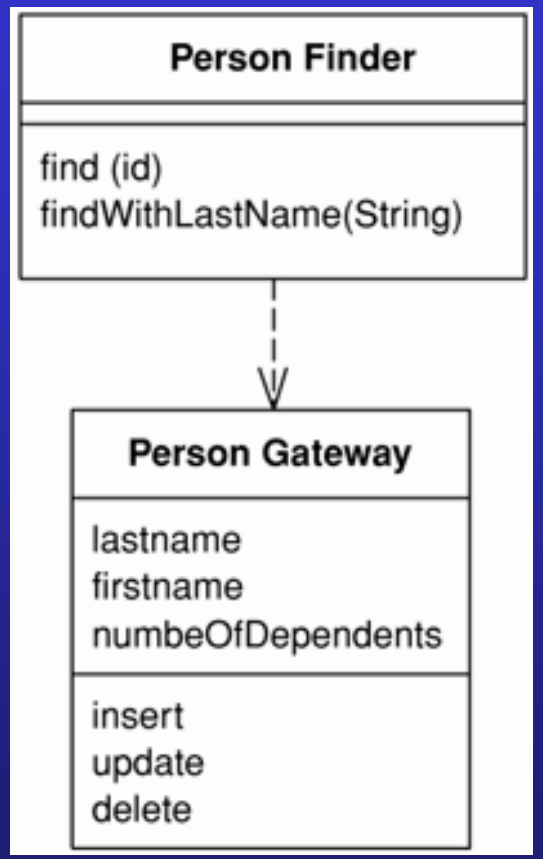


表数据入口

```
class PersonGateway...

      public IDataReader FindAll() {
         String sql = "select * from person";
         return new OleDbCommand(sql, DB.Connection).ExecuteReader();
      }
      public IDataReader FindWithLastName(String lastName) {
         String sql = "SELECT * FROM person WHERE lastname = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add(new OleDbParameter("lastname", lastName));
         return comm.ExecuteReader();
      }
      public IDataReader FindWhere(String whereClause) {
         String sql = String.Format("select * from person where {0}", whereClause);
         return new OleDbCommand(sql, DB.Connection).ExecuteReader();
      }
```



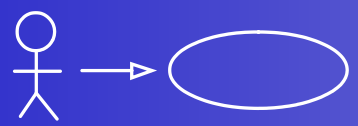
行数据入口



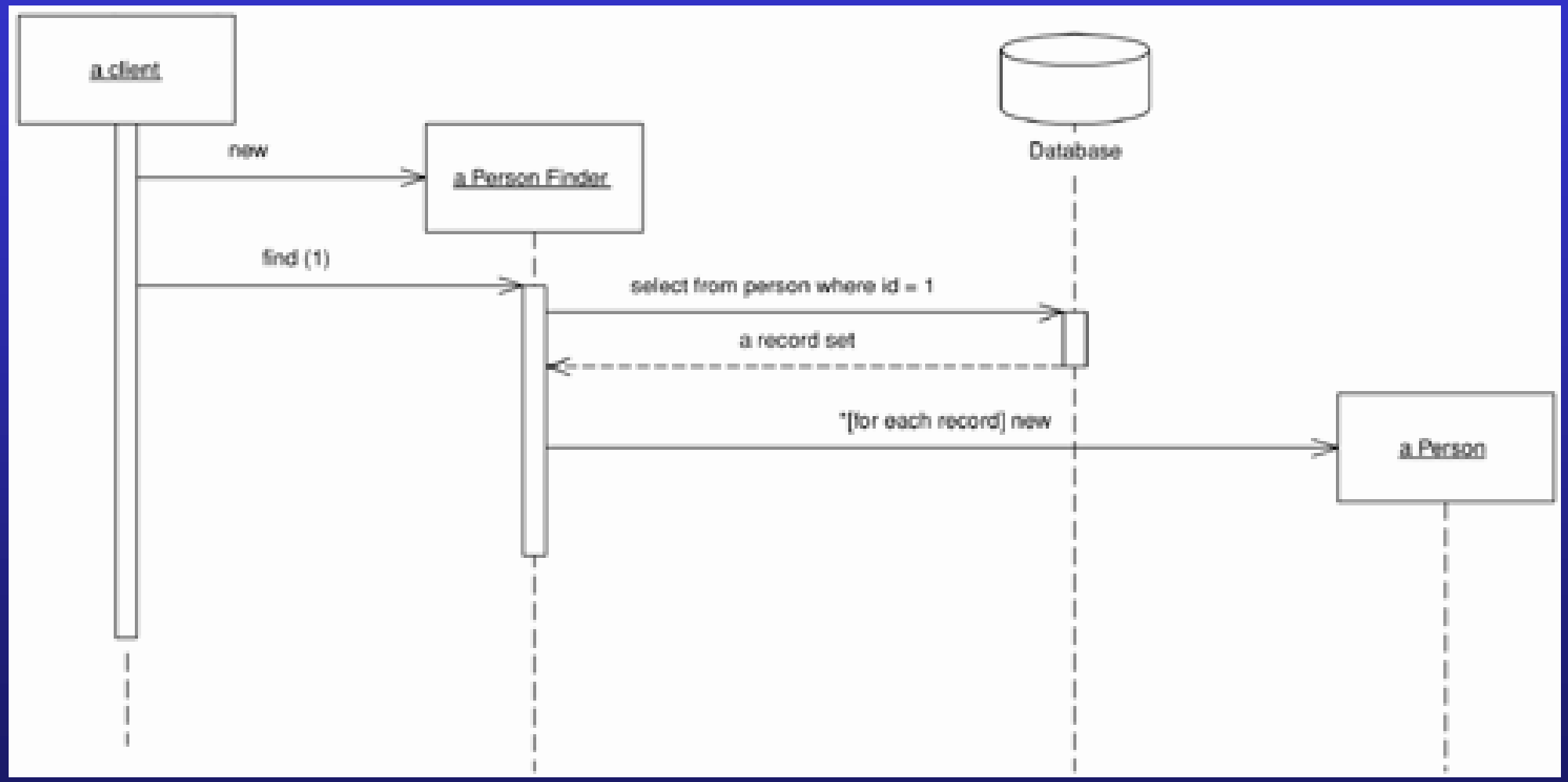
一行一个实例

行数据入口

- 把所有和一张表有关的SQL放在同一个类
- 针对每行有一个对象
- 增加/移除字段容易
- 通常和事务脚本一起使用
- 方法只包含SQL



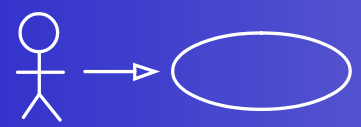
行数据入口


a client
Database
new
a Person Finder
find (1)
select from person where id = 1
a record set
*[for each record] new
a Person

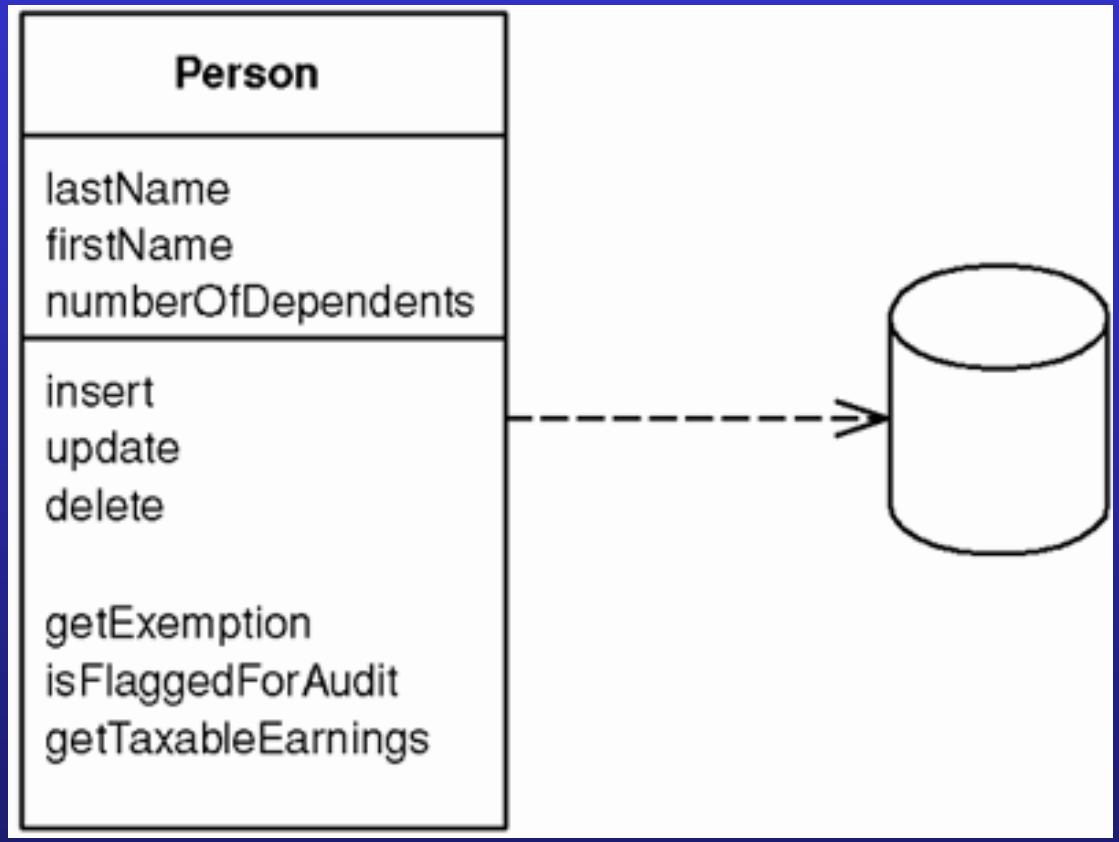
行数据入口

```
class PersonGateway...

   private static final String updateStatementString =
         "UPDATE people " +
         "  set lastname = ?, firstname = ?, number_of_dependents = ? " +
         "  where id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
         updateStatement.setInt(3, numberOfDependents);
         updateStatement.setInt(4, getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(updateStatement);
      }
   }
```

更新



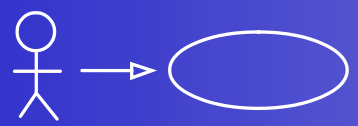
活动记录



领域模型＋数据逻辑

活动记录

- 很象行数据入口，除了
  - 某些方法是领域逻辑
  - 某些变量不存储在数据库
  - 更多为领域模型设计，而不是为数据库
- 对象模型和数据模型紧耦合
- 对象模型和数据模型同构时适用

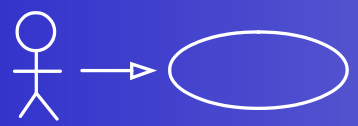


活动记录

```
class Person...

    private String lastName;
    private String firstName;
    private int numberOfDependents;
```

```
create table people (ID int primary key, lastname varchar,
                     firstname varchar, number_of_dependents int)
```

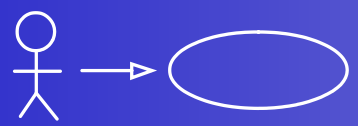
类结构和表结构一致



活动记录

```
class Person...

   private final static String updateStatementString =
         "UPDATE people" +
         "  set lastname = ?, firstname = ?, number_of_dependents = ?" +
         "  where id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
         updateStatement.setInt(3, numberOfDependents);
         updateStatement.setInt(4, getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

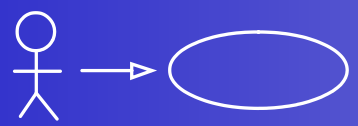
更新



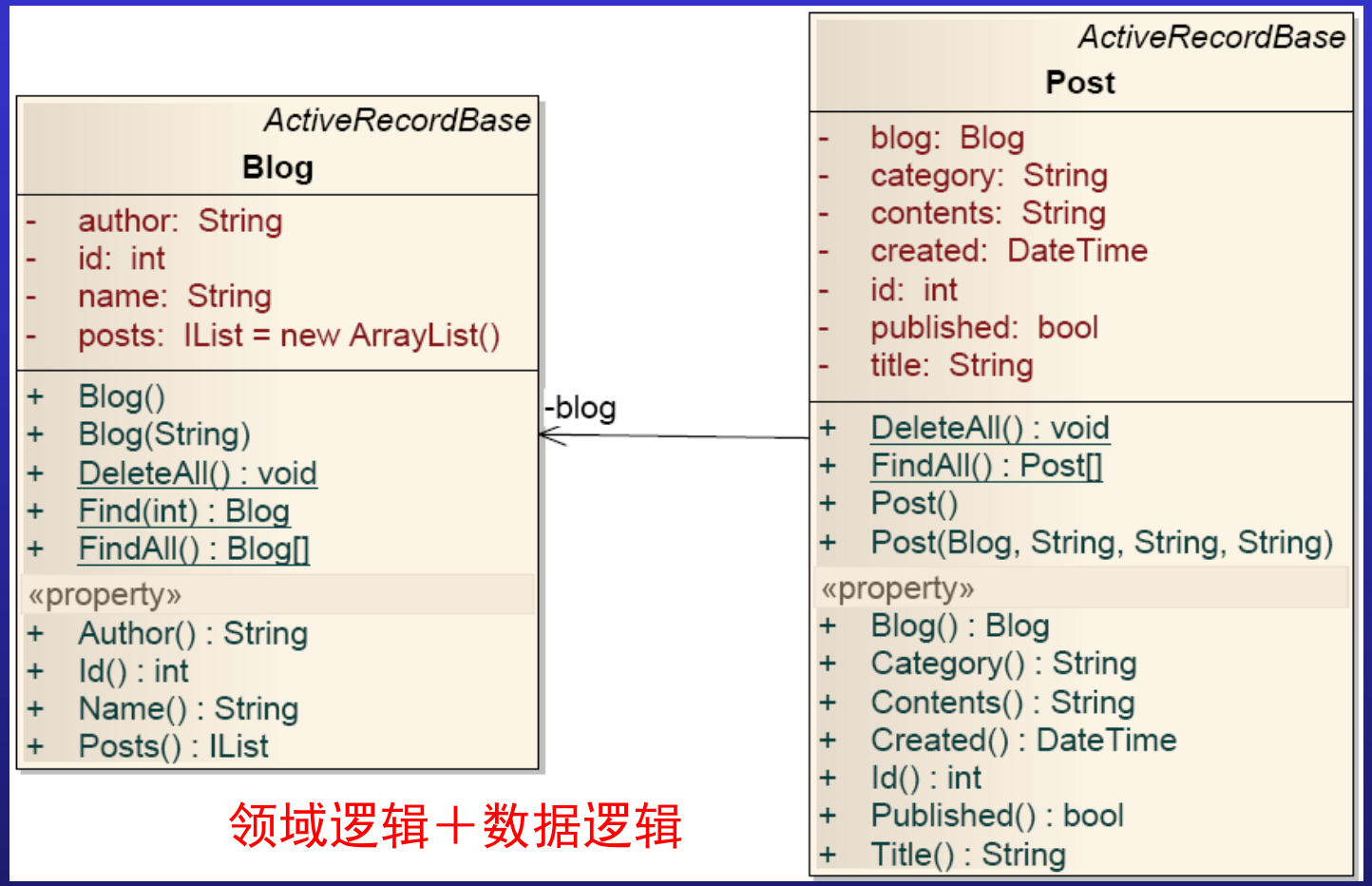
活动记录

```
class Person...

    public Money getExemption() {
        Money baseExemption = Money.dollars(1500);
        Money dependentExemption = Money.dollars(750);
        return baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents()));
    }
```

业务逻辑也放在一起

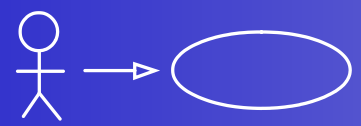


活动记录

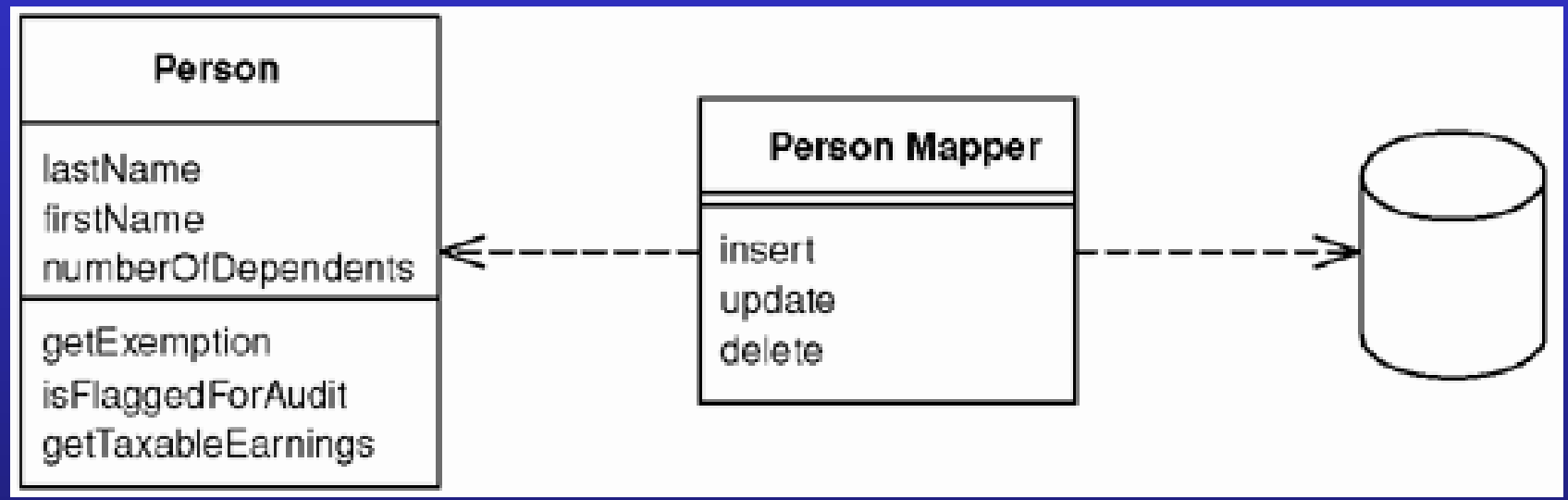


领域逻辑＋数据逻辑

分层超类型



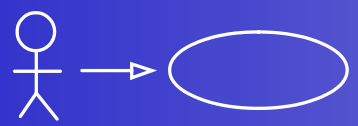
数据映射器



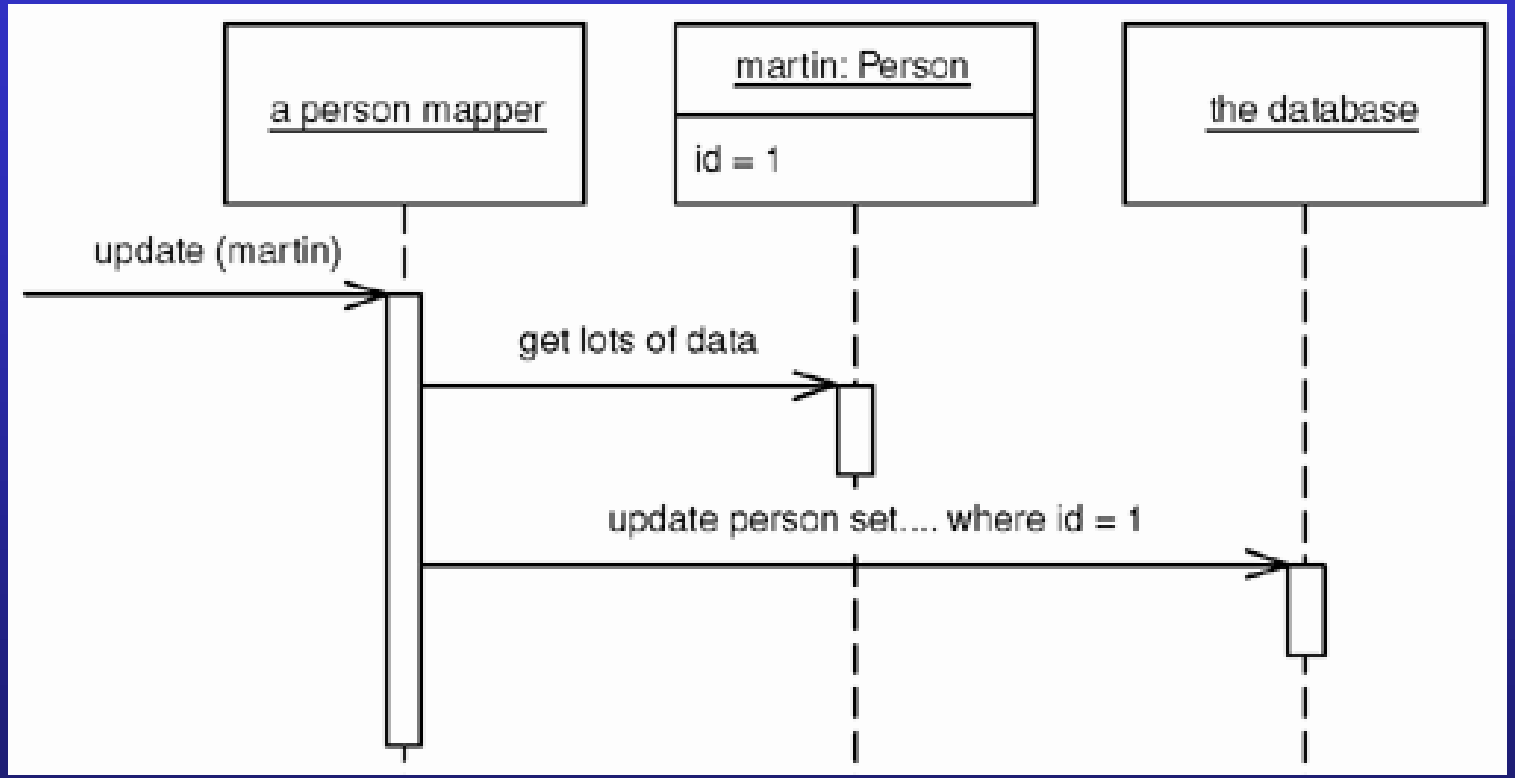
保持对象和数据库独立

数据映射器

- 在领域模型和数据库之间移动数据
- 领域模型基本不知道数据映射器
- 数据库设计独立于领域模型
- 和领域模型一起使用



数据映射器

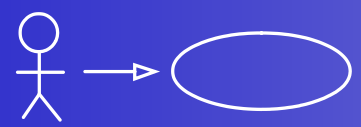

a person mapper
martin: Person
id = 1
the database
update (martin)
get lots of data
update person set.... where id = 1

更新

数据映射器

```
class PersonMapper...

   private static final String updateStatementString =
         "UPDATE people " +
         "  SET lastname = ?, firstname = ?, number_of_dependents = ? " +
         "  WHERE id = ?";
   public void update(Person subject) {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, subject.getLastName());
         updateStatement.setString(2, subject.getFirstName());
         updateStatement.setInt(3, subject.getNumberOfDependents());
         updateStatement.setInt(4, subject.getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

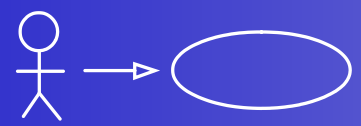
更新



数据映射器

- 提供一个对象负责协调
- NH中的Session
- EF中的ObjectContext

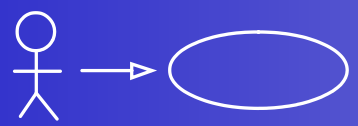

```
using (var context = new QuizEntities())
{
    var questiontype = new QuestionType() {Name="选择题"};
    context.QuestionTypeSet.AddObject(questiontype);
    context.SaveChanges();
}
```



业务层模式

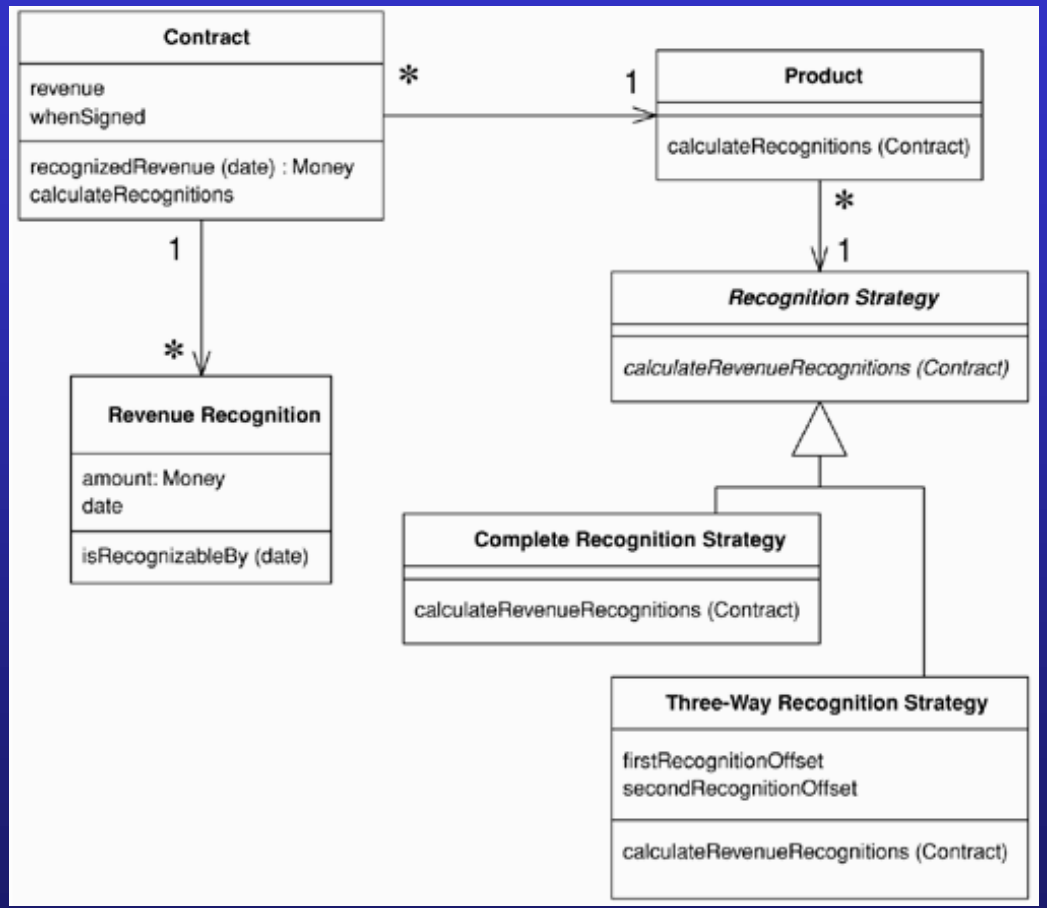
- 领域模型
- 表模块
- 事务脚本

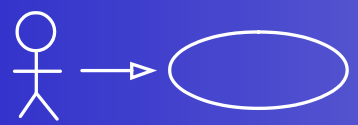
封装业务逻辑（无SQL）



领域模型

- 对象模型包含行为和数据
- 更OO的道路
- 每个领域对象对自己的状态和函数负责
- 两种风格
  - 简单（直接映射到DB）
  - 丰富（使用其他模式维护与其他对象的复杂关系）



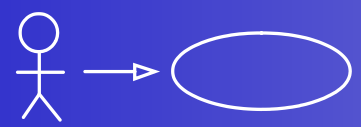


领域模型

```
public class OrderService
{
    public void DoOrderStuff(int orderId)
    {
        var order = OrderRepository.Get(orderId);

        order.DoOrderStuff();

        OrderRepository.Save(order);
    }
}
```

```
public class OrderDetail
{
    public int OrderDetailId { get; set; }
    public string Information { get; set; }

    public void DoStuff()
    {
        // Do clever stuff
    }
}
```

```
public class Order
{
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public DateTime ShipDate { get; set; }
    public int CustomerId { get; set; }
    public int EmployeeId { get; set; }
    public IList<OrderDetail> OrderDetails { get; set; }

    public bool IsOrderValid()
    {
        return (OrderDate == DateTime.Today);
    }

    public void DoOrderStuff()
    {
        foreach (var orderDetail in OrderDetails)
        {
            orderDetail.DoStuff();
        }
    }
}
```



领域

```
public class OrderService
{
    public void DoOrderStuff(int orderId)
    {
        var order = OrderRepository.Get(orderId);

        OrderHandler.DoOrderStuff(order);

        OrderRepository.Save(order);
    }
}
```

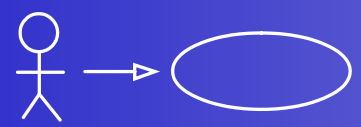
```
public class Order
{
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public DateTime ShipDate { get; set; }
    public int CustomerId { get; set; }
    public int EmployeeId { get; set; }
    public IList<OrderDetail> OrderDetails { get; set; }

    public bool IsOrderValid()
    {
        return (OrderDate == DateTime.Today);
    }
}
```

```
public static class OrderHandler
{
    public static void DoOrderStuff(Order order)
    {
        foreach (var orderDetail in order.OrderDetails)
        {
            DoOrderDetailStuff(orderDetail);
        }
    }

    public static void DoOrderDetailStuff(OrderDetail detail)
    {
        // business logic
    }
}
```

xxHelper, xxManager, xxHandler

```
public class OrderDetail
{
    public int OrderDetailId { get; set; }
    public string Information { get; set; }
}
```

贫血领域模型

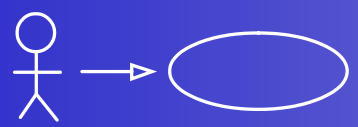


领域模型

```
public class OrderService
{
    public void DoOrderStuff(int orderId)
    {
        var order = Order.Get(orderId);

        order.DoOrderStuff();

        order.Save();
    }
}
```

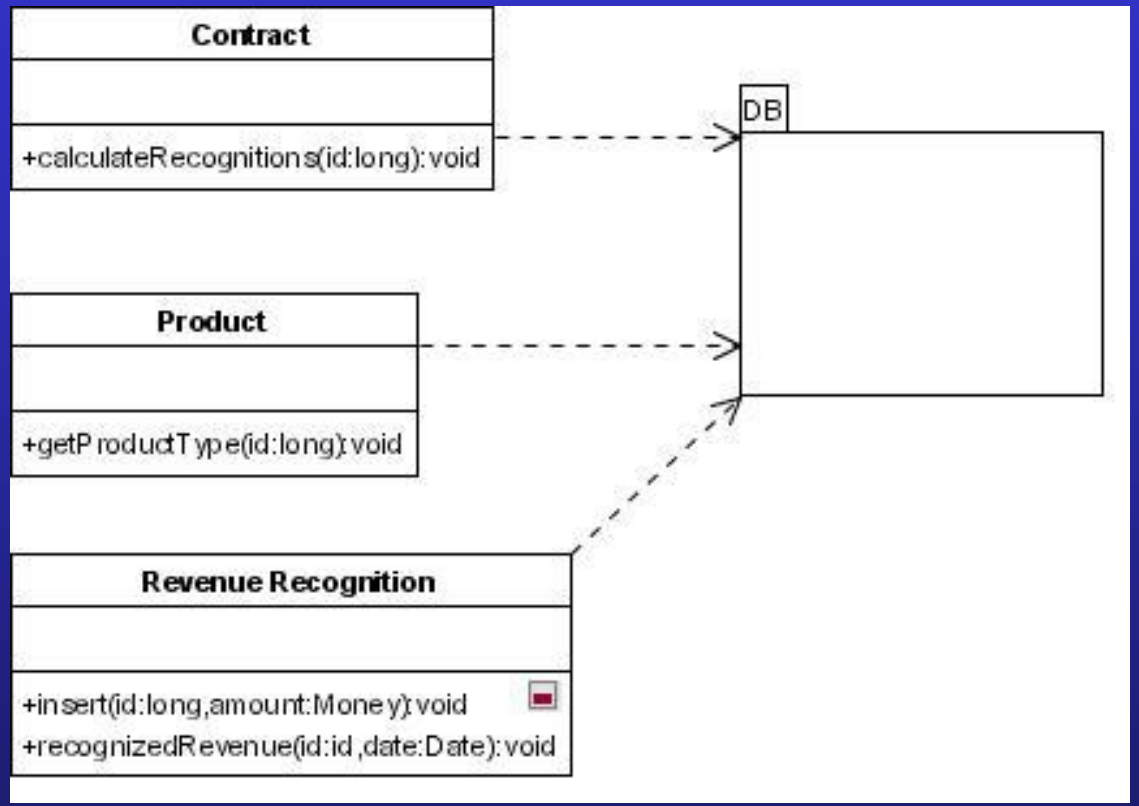
```
[ActiveRecord]
public class Order
{
    public Order() { }

    public Order(int orderId) { ... }

    [PrimaryKey]
    public int OrderId { get; set; }
    [Property]
    public DateTime OrderDate { get; set; }
    [Property]
    public DateTime ShipDate { get; set; }
    [Property]
    public string CustomerId { get; set; }
    [Property]
    public int EmployeeId { get; set; }
    [HasMany]
    public IList<OrderDetail> OrderDetails { }

    public void Insert()
    {
        //Insert record in table
    }

    public void Delete()
    {
        //Delete the record from the table
    }

    public static int GetOrdersCount()
    {
        //Return the number of orders in the table
    }

    public bool IsOrderValid()
    {
        return (OrderDate == DateTime.Today);
    }
```

活动记录

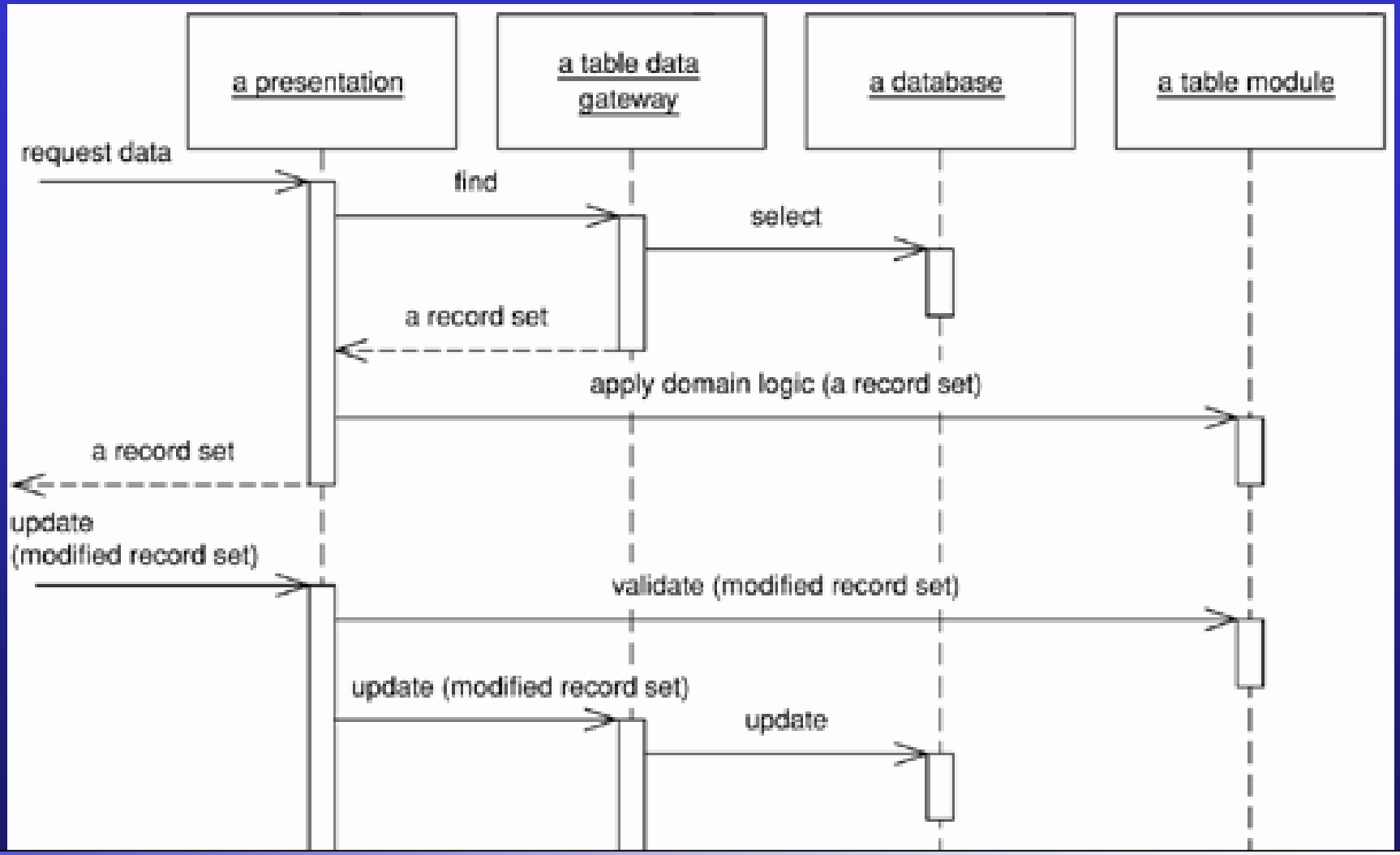


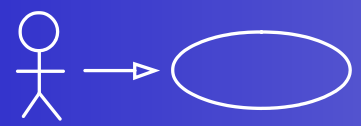
表模块

- 单个对象代表一张数据库表或视图
  - 针对对象操作变复杂
    - aTModule.getadress(long empid)
- 对象映射更象数据库表
- 通常返回记录集



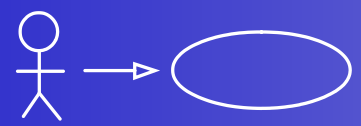
表模块


a presentation
a table data gateway
a database
a table module
request data
find
select
a record set
apply domain logic (a record set)
a record set
update
(modified record set)
validate (modified record set)
update (modified record set)
update



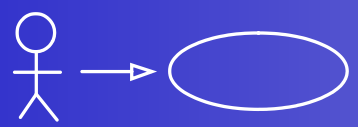
表模块

```
class Contract...

      public void CalculateRecognitions (long contractID) {
         DataRow contractRow = this[contractID];
         Decimal amount = (Decimal)contractRow["amount"];
         RevenueRecognition rr = new RevenueRecognition (table.DataSet);
         Product prod = new Product(table.DataSet);
         long prodID = GetProductId(contractID);
         if (prod.GetProductType(prodID) == ProductType.WP) {
            rr.Insert(contractID, amount, (DateTime) GetWhenSigned(contractID));
         }else if (prod.GetProductType(prodID) == ProductType.SS) {
            Decimal[] allocation = allocate(amount,3);
            rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
            rr.Insert(contractID, allocation[1], (DateTime) GetWhenSigned(contractID).
➥ AddDays(60));
            rr.Insert(contractID, allocation[2], (DateTime) GetWhenSigned(contractID).
➥ AddDays(90));
         }else if (prod.GetProductType(prodID) == ProductType.DB) {
            Decimal[] allocation = allocate(amount,3);
            rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
            rr.Insert(contractID, allocation[1], (DateTime) GetWhenSigned(contractID).
➥ AddDays(30));
            rr.Insert(contractID, allocation[2], (DateTime) GetWhenSigned(contractID).
➥ AddDays(60));
         }else throw new Exception("invalid product id");
      }
```



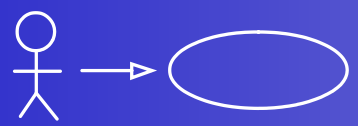
表模块

```
public class OrdersManager
{
    private DataSet _data;
    public OrdersManager(DataSet data)
    {
        _data = data;
    }

    public DataTable Orders
    {
        get { return _data.Tables[0]; }
    }

    public DataRow GetRow(int index)
    {
        return _data.Tables[0].Rows[index];
    }

    public DataRow GetRowById(int orderId)
    {
        return _data.Tables[0].Select(...);
    }

    public int Update(DataRow row) { ... }

    public int Insert(DataRow row) { ... }
}
```

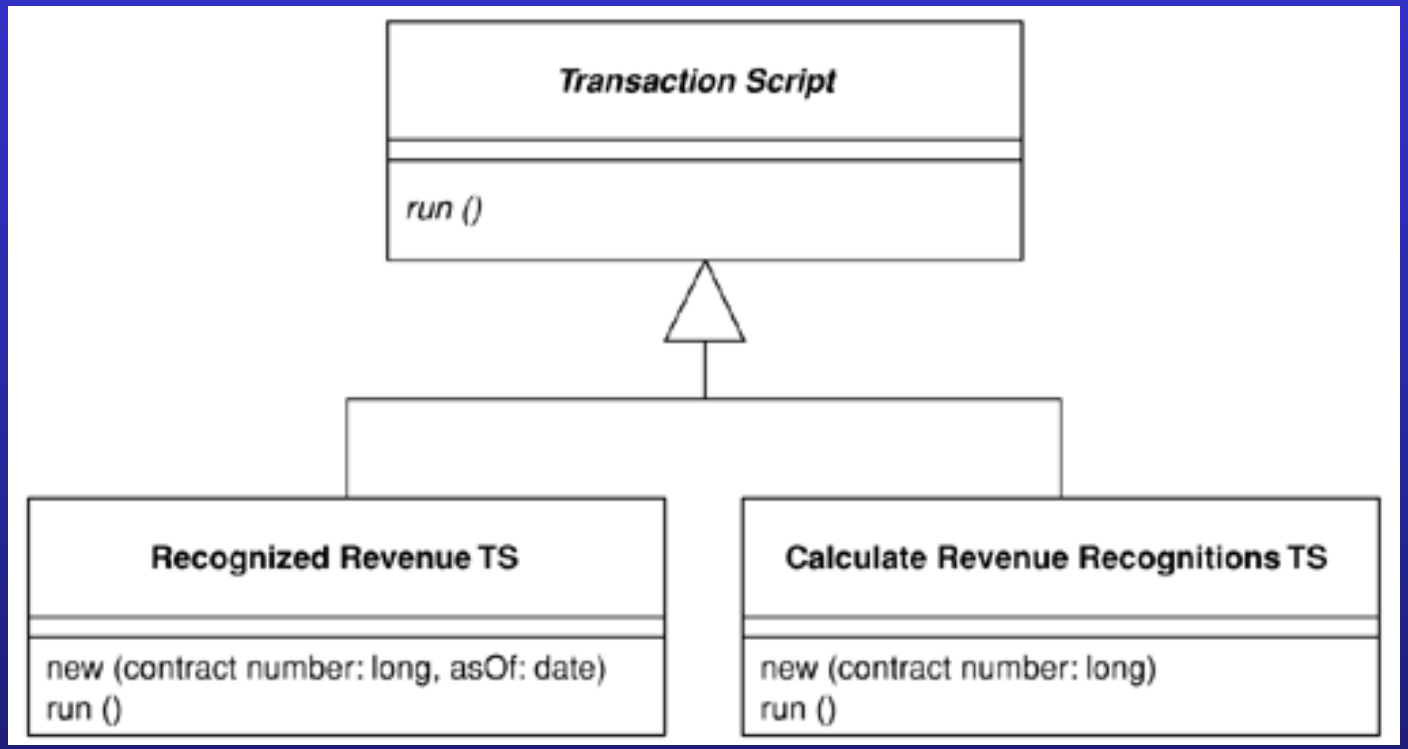


事务脚本

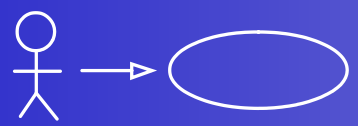
- 操作按过程调用组织
- 可使用Command和/或Strategy模式实现
- 操作有明显的边界



事务脚本

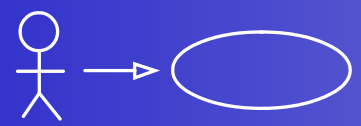


可以组织到类中



事务脚本

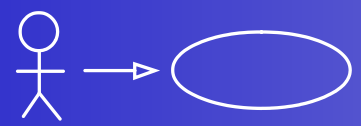
```
class RecognitionService...

    public Money recognizedRevenue(long contractNumber, MfDate asOf) {
        Money result = Money.dollars(0);
        try {
            ResultSet rs = db.findRecognitionsFor(contractNumber, asOf);
            while (rs.next()) {
                result = result.add(Money.dollars(rs.getBigDecimal("amount")));
            }
            return result;
        }catch (SQLException e) {throw new ApplicationException (e);
        }
    }
```



事务脚本

```
public class OrderService
{
    public void DoOrderStuff(int orderId)
    {
        // Retrieve order
        // Check status of order
        // Do something with order
        // Do something else
        // Commit the transaction
    }
}
```

纯面向过程



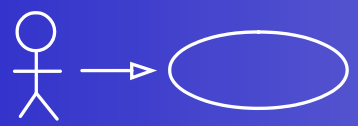
实体

- 生命周期中保持连续性
- 独立于其属性

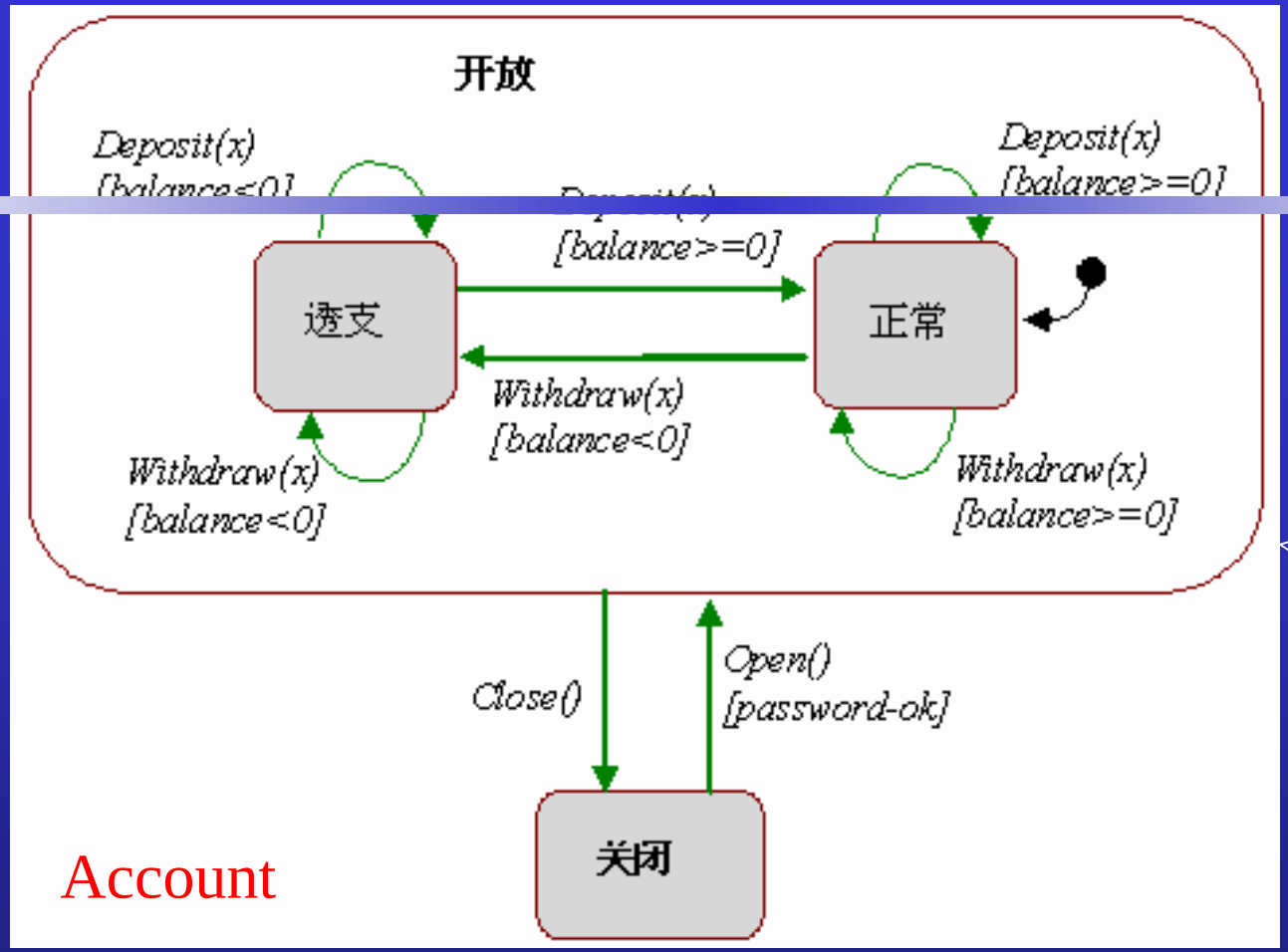
- 可以是人、城市、汽车、彩票、银行交易…

- 关注它的状态对系统有意义吗？

	座位
对号入座	实体
开放入座	值对象



实体



其实每个类
都有状态

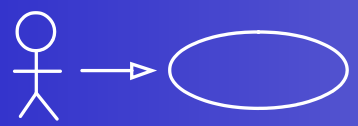
行为表现不同
属性值的分组

区分 “有状态” 和 “无状态”

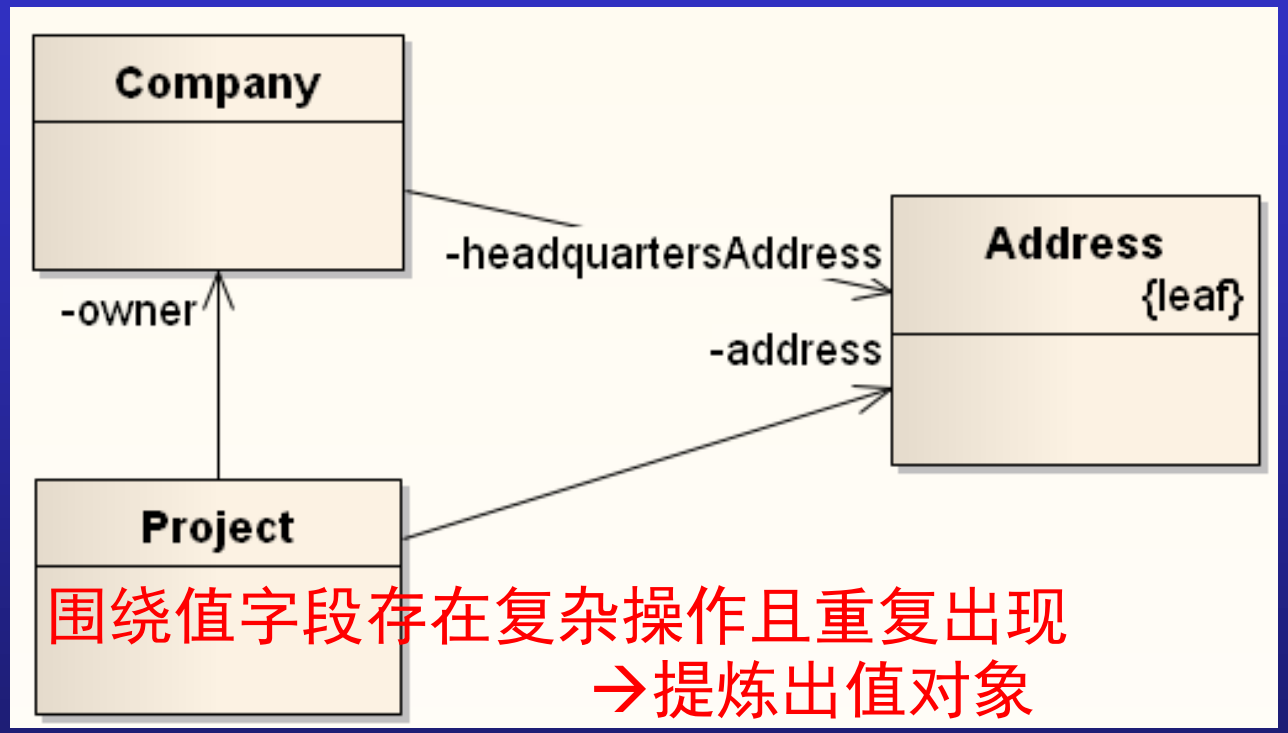
实体

- 状态多→事物→实体→责任起点→聚合的根
- 订单是核心，还是商品是核心？
- 良好状态机——类的完备性

哪些类有丰富的状态？



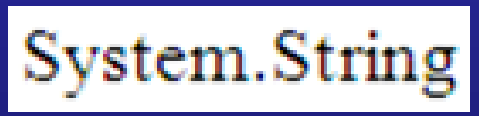
值对象

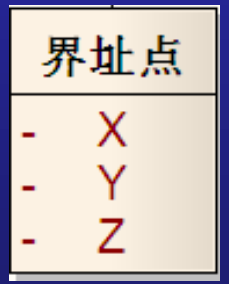

Company
-headquartersAddress
Address
{leaf}
-owner
-address
Project
围绕值字段存在复杂操作且重复出现
→提炼出值对象

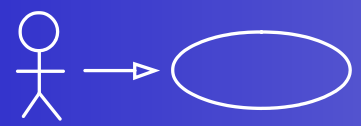

<<值对象>>
通信地址
地址
端口
。。。


<<值对象>>
用电量
总。。
峰。。
谷。。
。。。


<<值对象>>
时间
年
月
日
时
分


System.String


界址点
- X
- Y
- Z

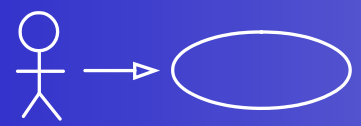


值对象

```
class CustForm extends ActionForm
   private String phone

class AddCustAction extends Action ... ...
   execute(...)
    String phone = form.getPhone();
    custserv.addCust(..., phone, ...);

class CustomerServiceBean ...
   void addCust(..., String phone, ...)
   throws ValidationException
       ...
       if (!justnumbers(phone)) ...
          throw new ValidationException();
       ...
       dbstmt.setString(4, phone);

   static boolean justnumbers(String s) ...
```

```
SalesRep findSalesRepresentative(String phone) {
  // phone directly assoc with sales rep?
  Object directrep = phone2repMap.get(phone);
  if (directrep != null)
    return (SalesRep) directrep;

  // find area code
  String prefix = null;
  for (int i=0; i<phone.length(); i++){
    String begin = phone.subString(0,i);
    if(isAreaCode(begin)) {
      prefix = begin;
      break;
    }
  }
  String areacode = prefix;

  // exists area representative?
  Object arearep = area2repMap.get(areacode);
  if (arearep != null)
    return (SalesRep) arearep;

  // neither direct nor area sales representative
  return null;
}
```

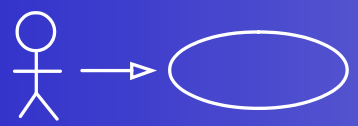
电话号码



值对象

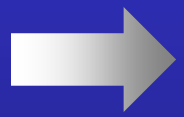
```
public class PhoneNumber {
   private final String number;

   public PhoneNumber(String number) {
    if(!isValid(number))
            throw …
            this.number = number;
   }

   public String getNumber() {
      return number;
    }

   static public boolean isValid(String number) {
      return number.matches("[0-9]*");
   }

   public String getAreaCode() {
     String prefix = null;
     for (int i=0; i< number.length(); i++){
       String begin = number.subString(0,i);
       if(isAreaCode(begin)) {
         prefix = begin;
         break;
     }
     return prefix;
   }

   private boolean isAreaCode(String prefix) { ... }
}
```

提炼



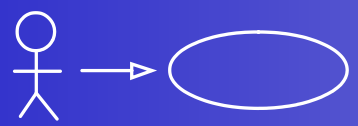
值对象

```
void addCust(String,
   String, String,
   int,
   int,
   String, String,
   boolean)
```

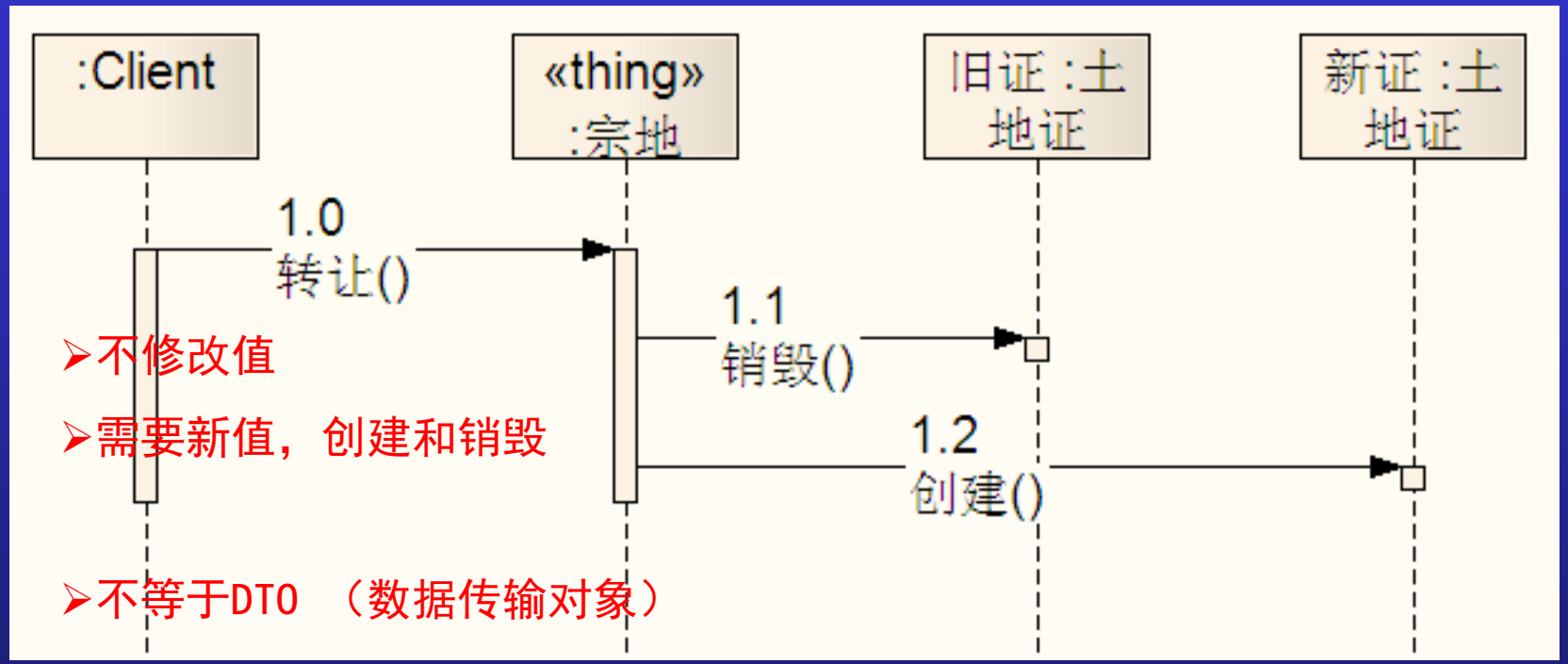


```
void addCust(Name,
   PhoneNumber, PhoneNumber,
   CreditStatus,
   SalesRepId,
   Name, PhoneNumber,
   ParnerStatus)
```

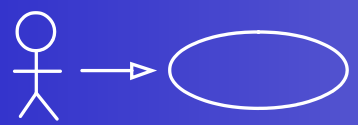
API更清晰，领域逻辑聚焦



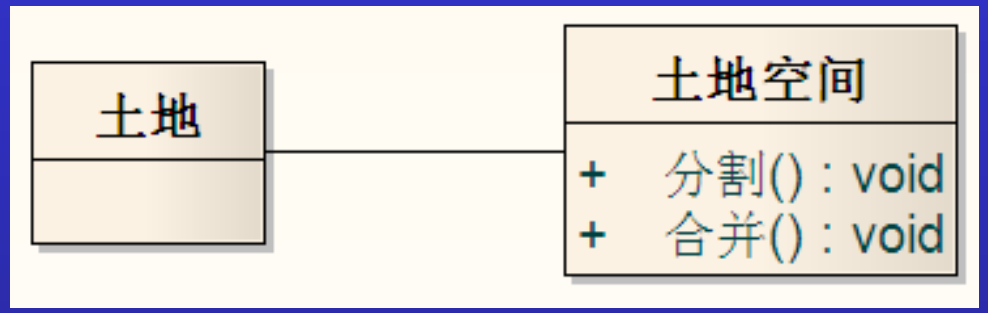
值对象

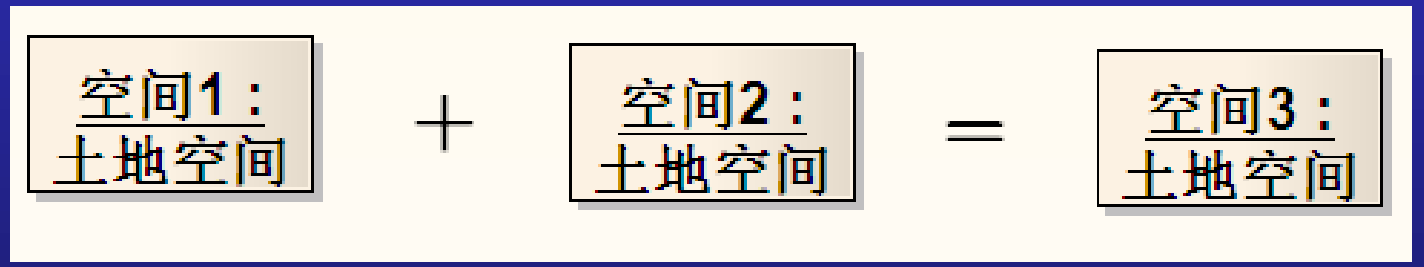


无状态，释放实体的复杂性



值对象

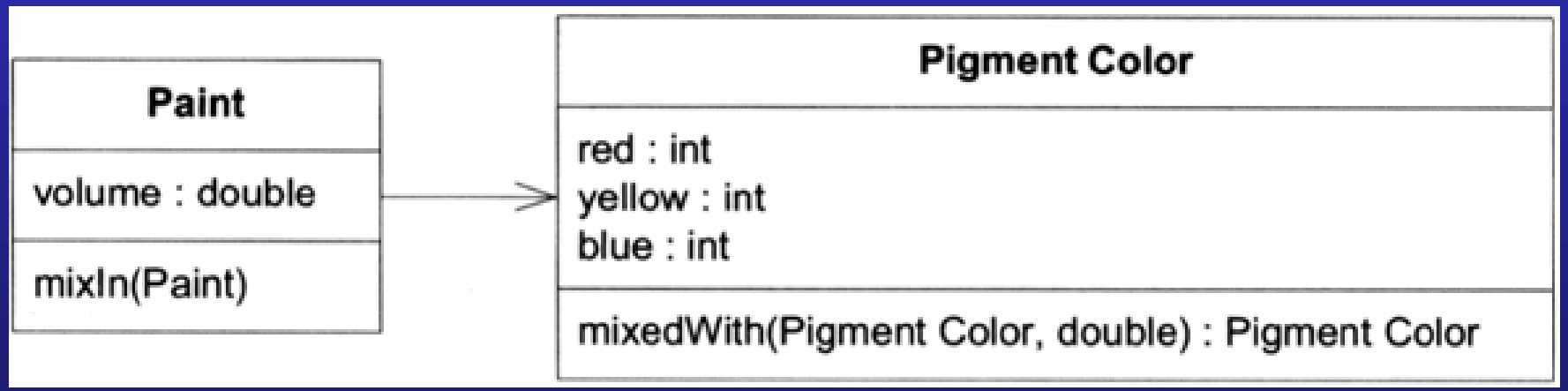




运算——无副作用操作

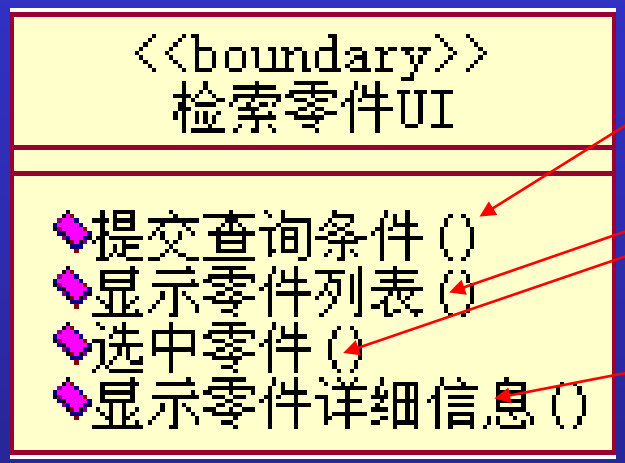
值对象

分解出颜料颜色值对象



颜色的运算

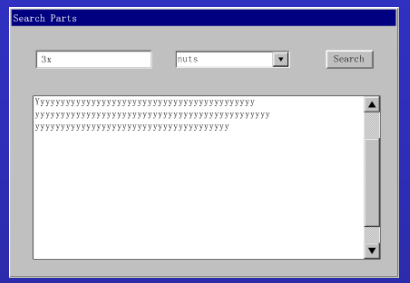
界面层

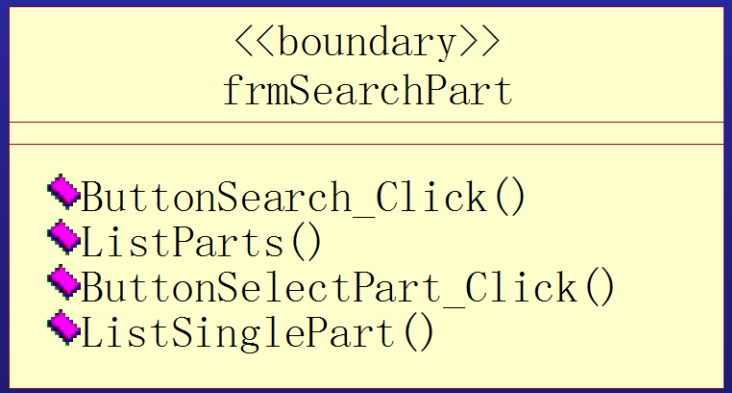


查询条件输入框，“提交”按钮

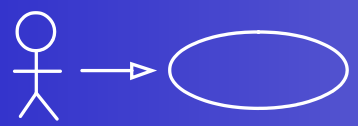
“列表框”

“零件详细信息”的表单

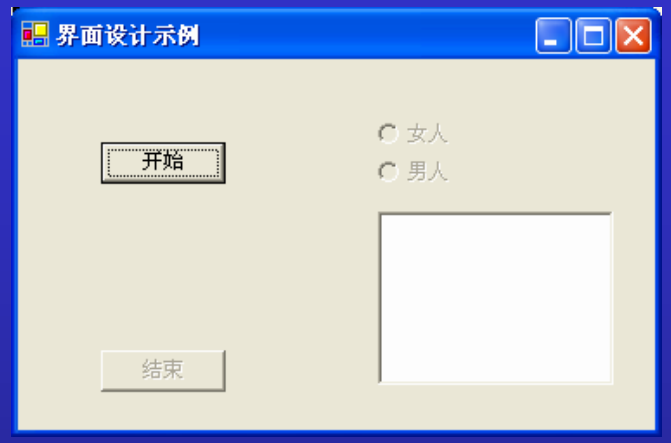


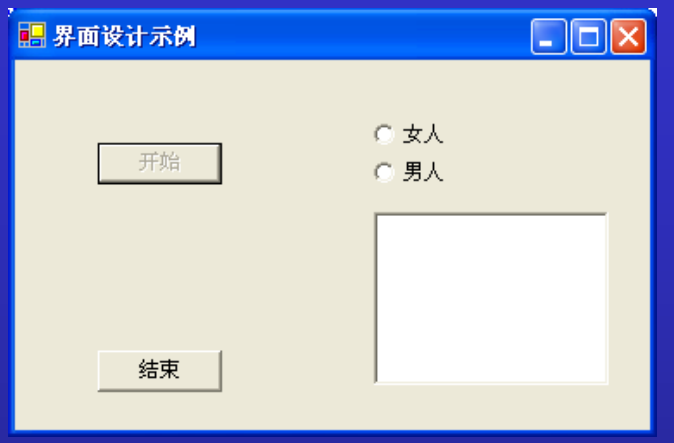


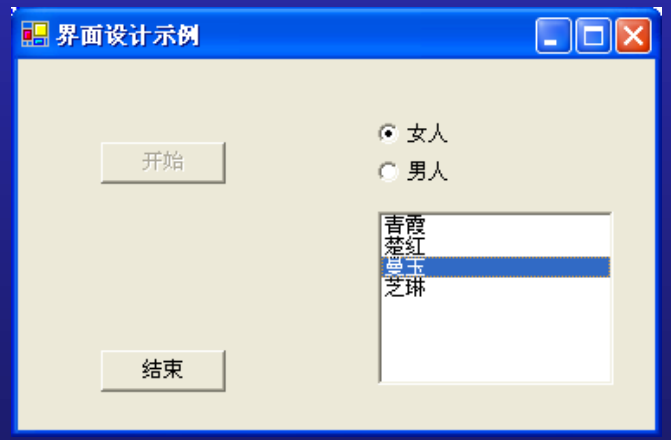
最简洁的界面：刚好能履行分析所赋予的责任

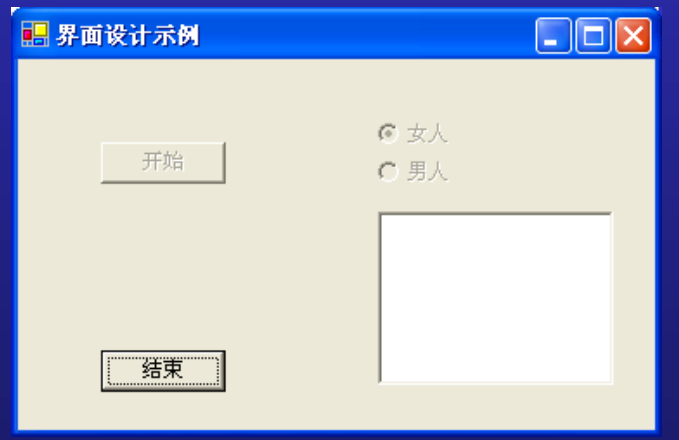


界面层

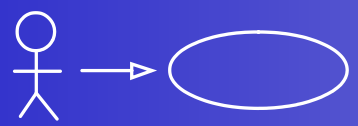




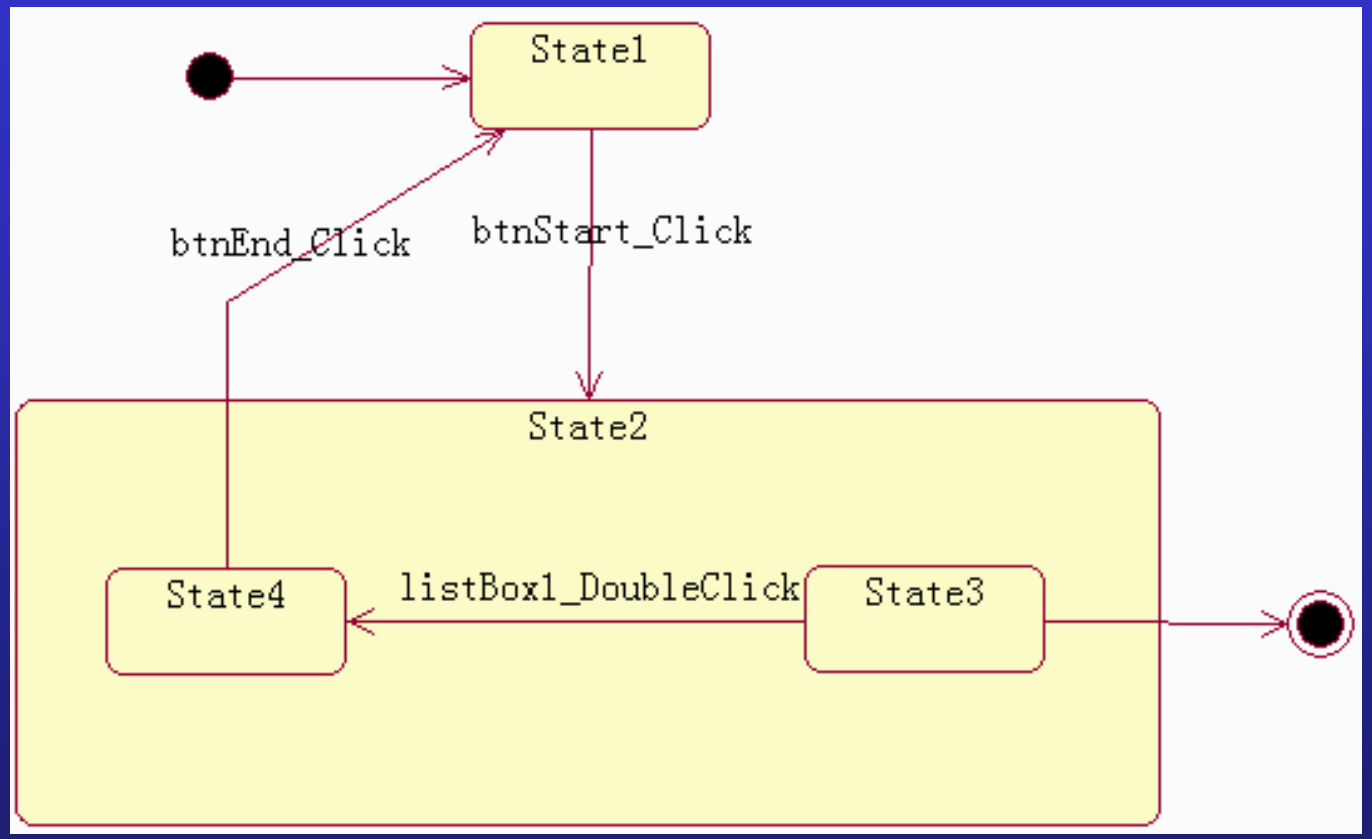




界面类接收大量消息，不断改变状态



界面层



以状态图主导界面设计

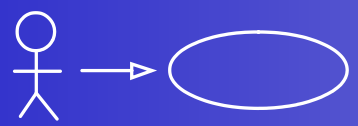
界面层

功能需求—>可靠性需求—>可用性需求

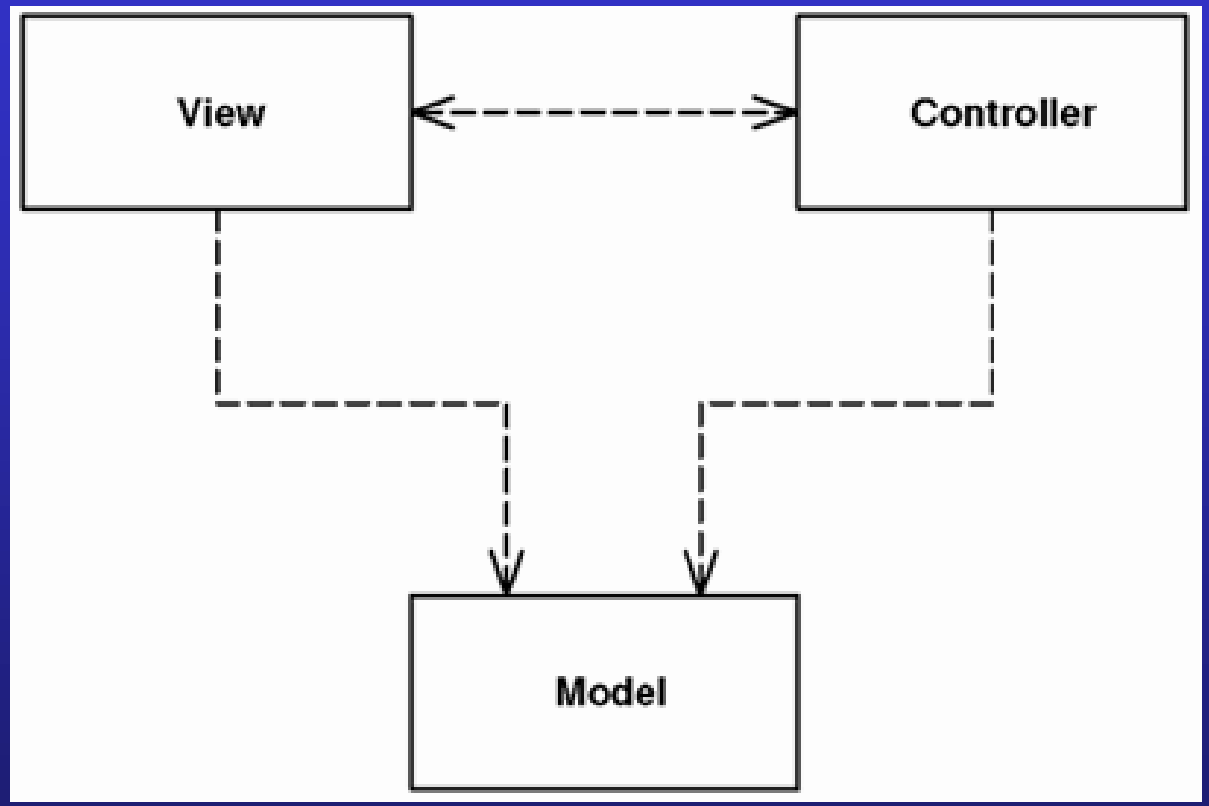


界面正确　　界面稳定　　界面高效

交互设计越来越重要

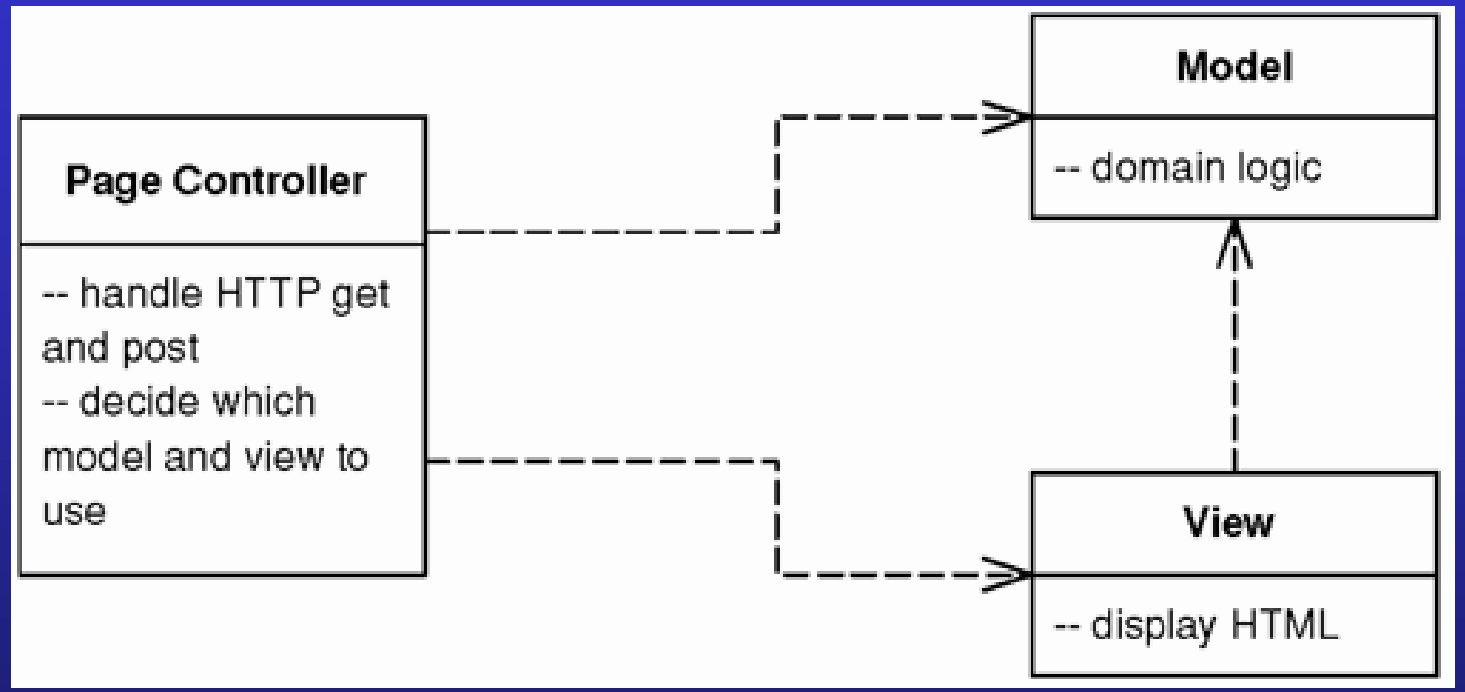


界面层



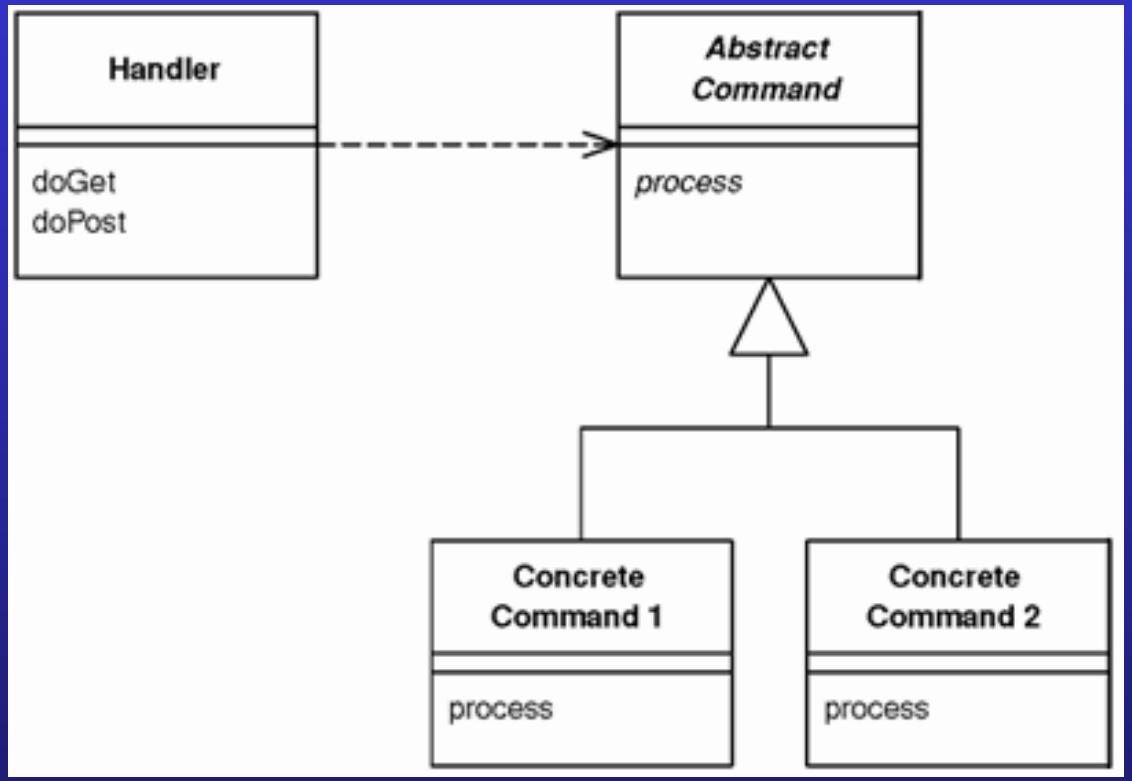
把用户界面交互分拆到三种不同角色

界面层

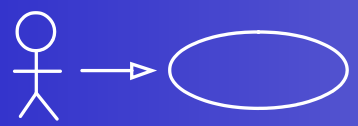


页面控制器——为每一个逻辑页面准备一个控

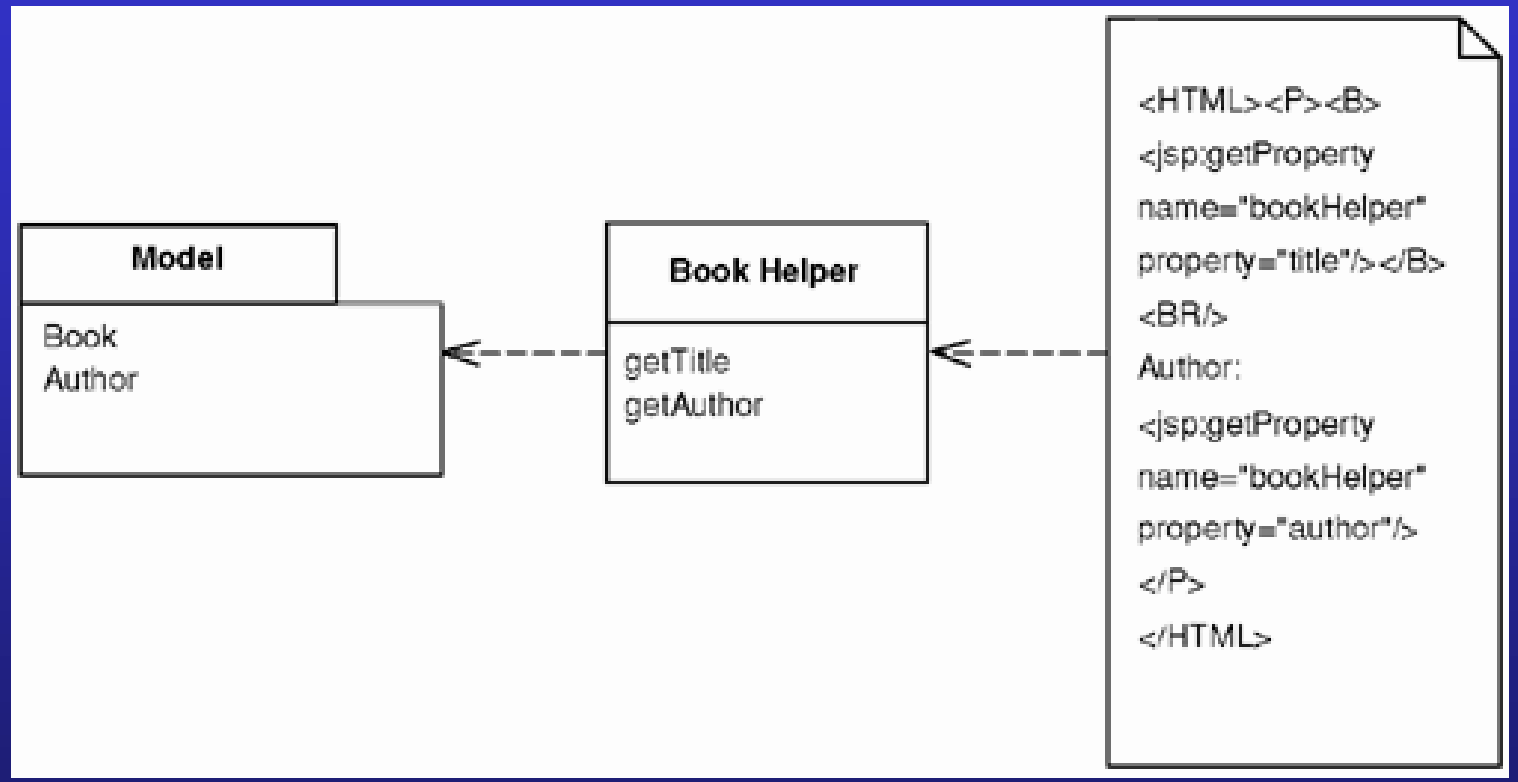
界面层



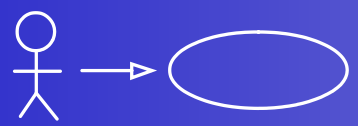
前端控制器——一个控制器处理站点所有请求



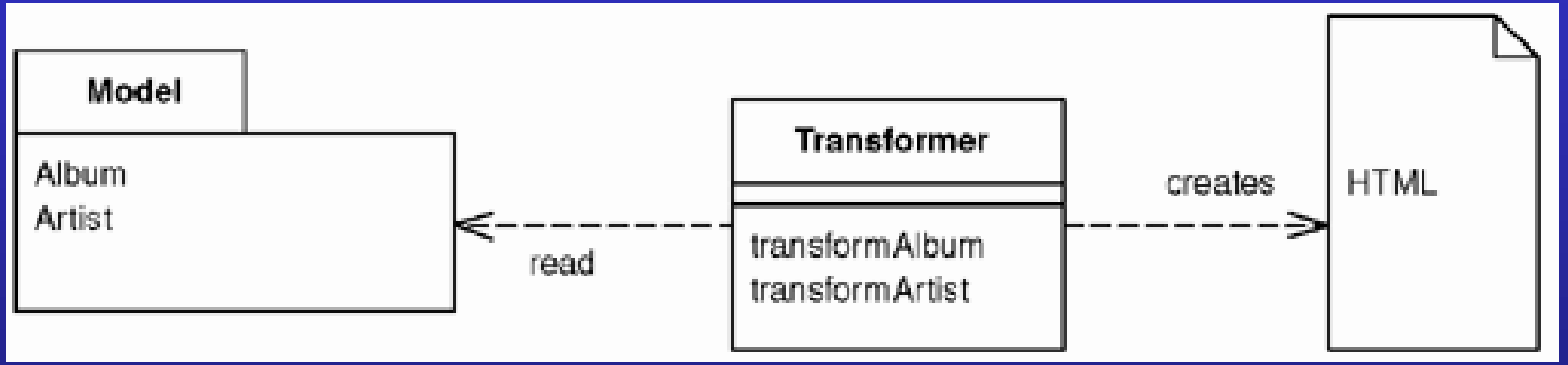
界面层



模板视图——在HTML页面嵌入标记

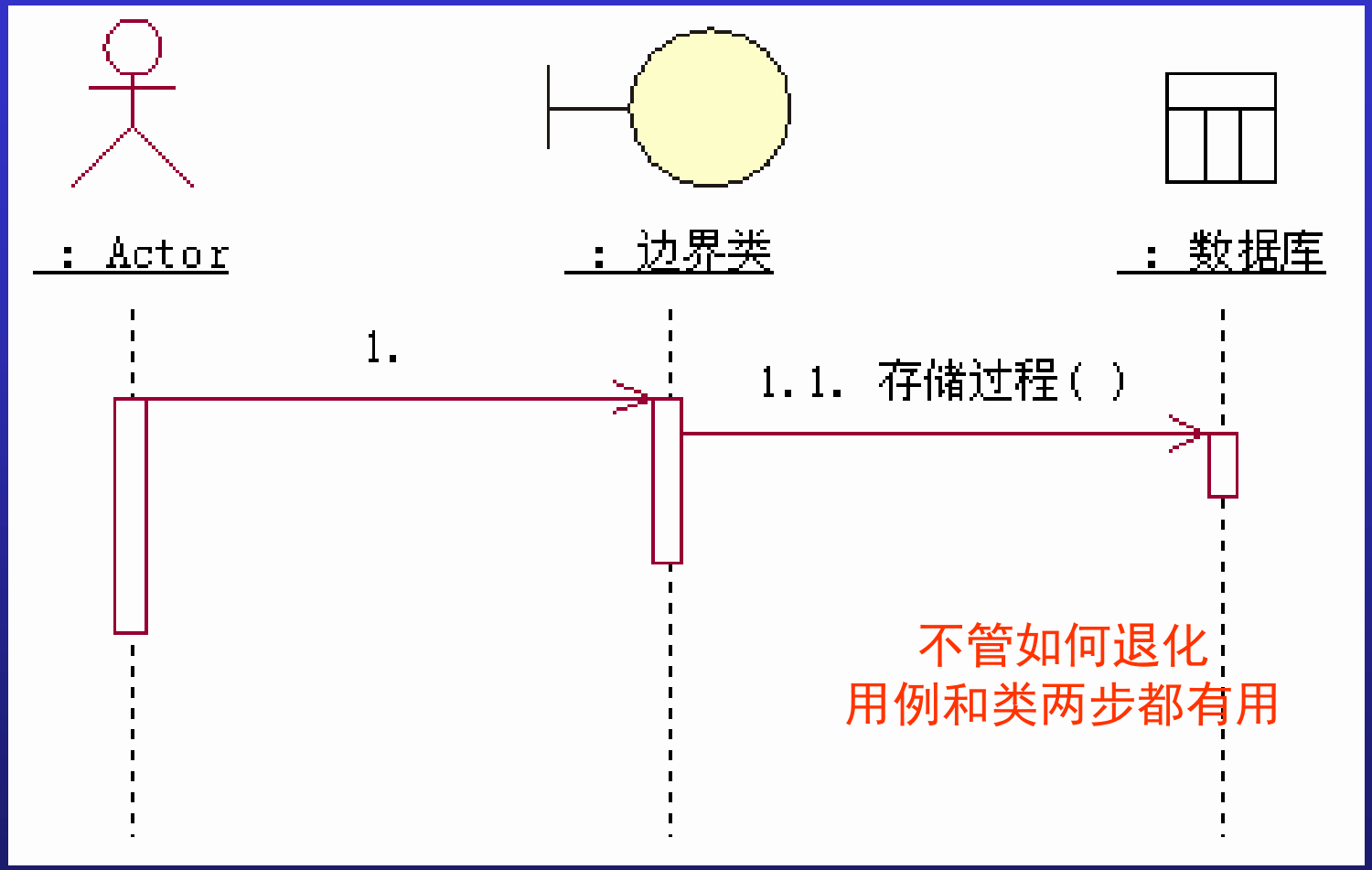


界面层



转换视图——逐项处理领域数据，把它们转成HTML

退化的实现

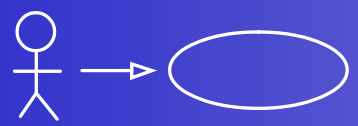


存储过程

退化的实现

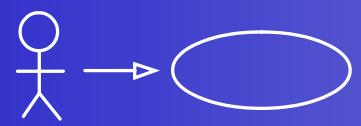
- 类、属性、关联→struct
- 方法→函数，把结构的指针作为参数
- 泛化（多态）→类描述器（VTable）维护

C实现

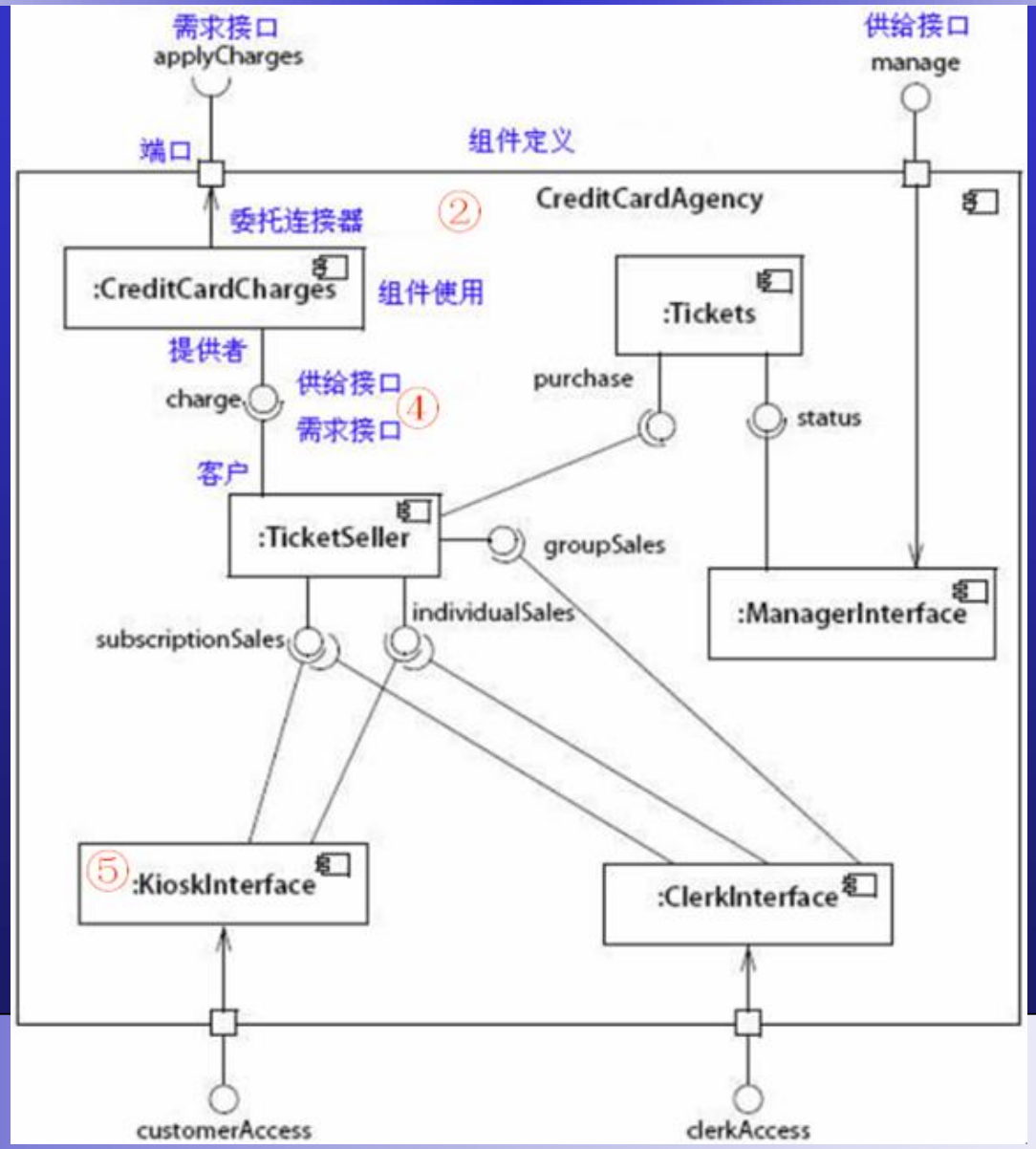


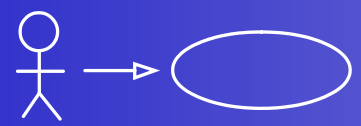
其他图

- 组件图（Component Diagram）—类的逻辑分包
- 部署图（Deployment Diagram）—组件的物理分布
- 包图—各种元素分组

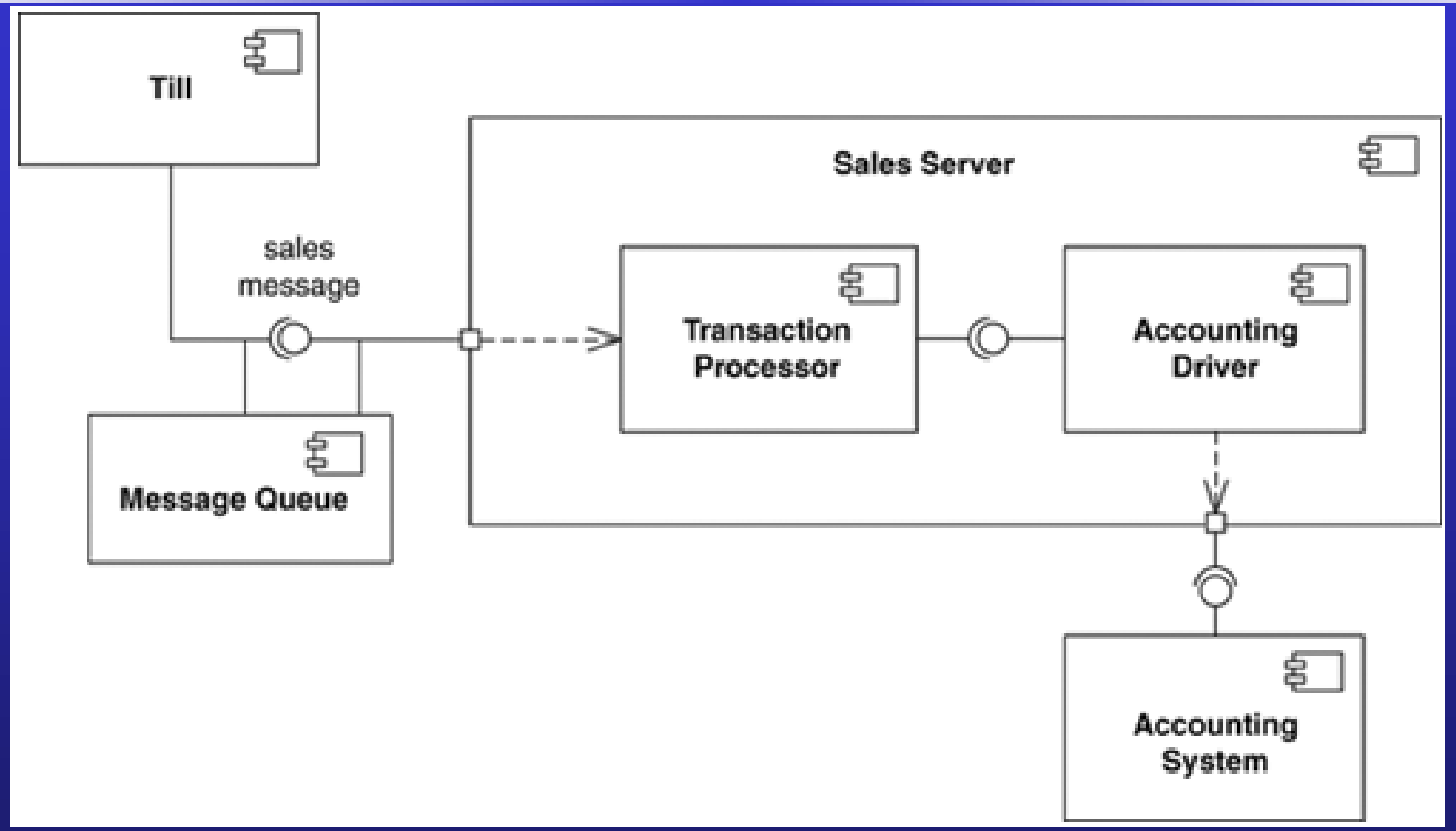


组件图


需求接口
applyCharges
供给接口
manage
端口
组件定义
CreditCardAgency
委托连接器
②
:CreditCardCharges
组件使用
:Tickets
提供者
供给接口
charge
④
需求接口
purchase
status
客户
:TicketSeller
groupSales
:ManagerInterface
individualSales
subscriptionSales
⑤
:KioskInterface
:ClerkInterface
customerAccess
clerkAccess

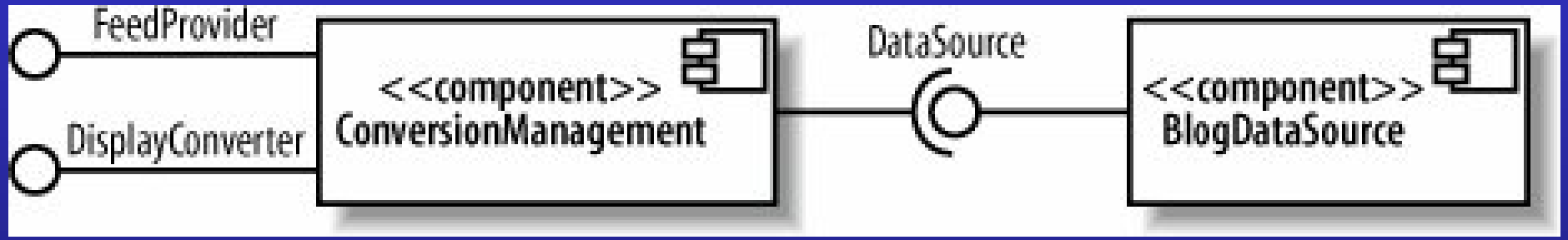


组件图


Till
Sales Server
sales
message
Transaction
Processor
Accounting
Driver
Message Queue
Accounting
System

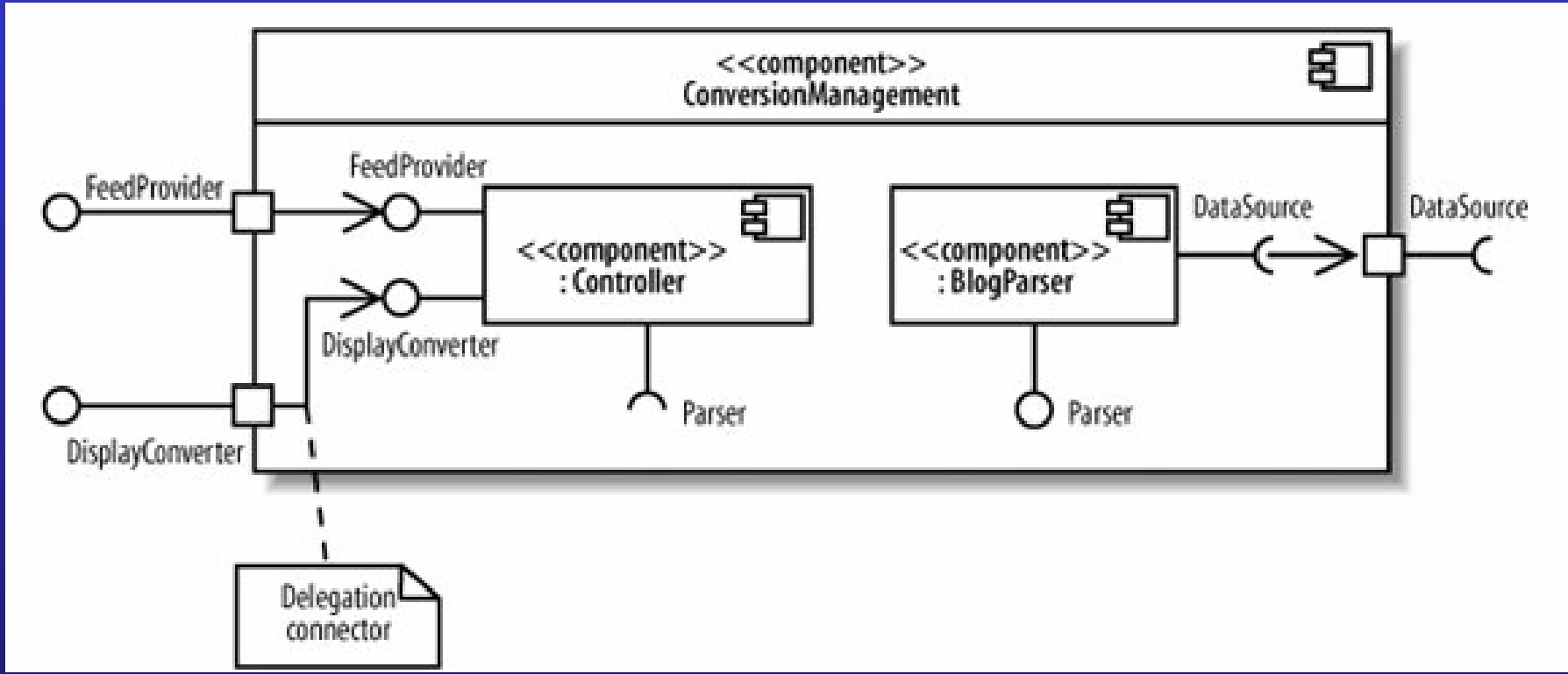
组件的装配

组件图


FeedProvider
DisplayConverter
<<component>>
ConversionManagement
DataSource
<<component>>
BlogDataSource

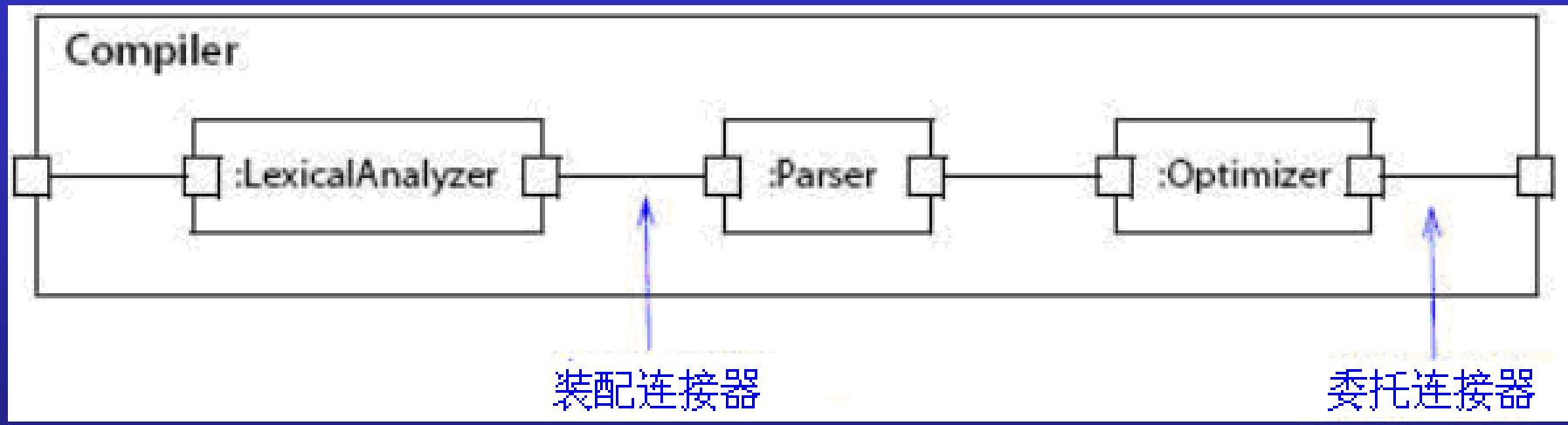
组件的装配

组件图



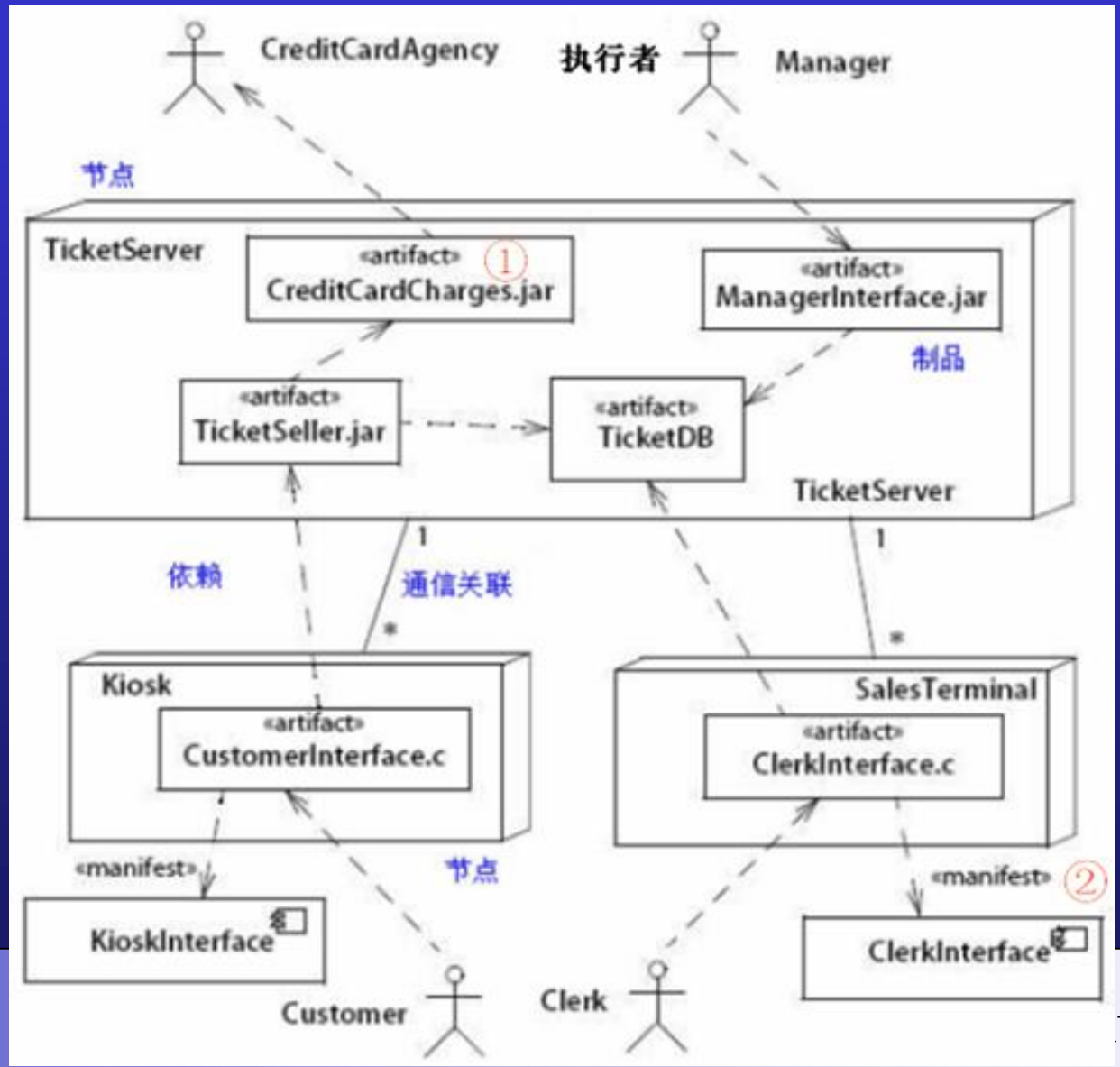
连接器一端口之间的桥梁

组件图

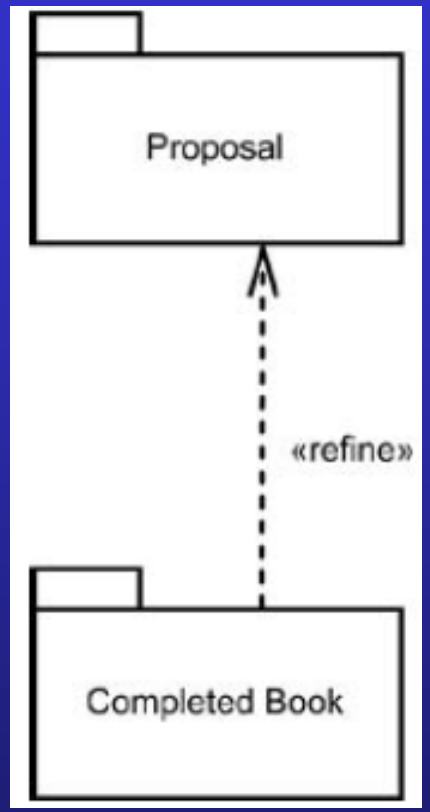


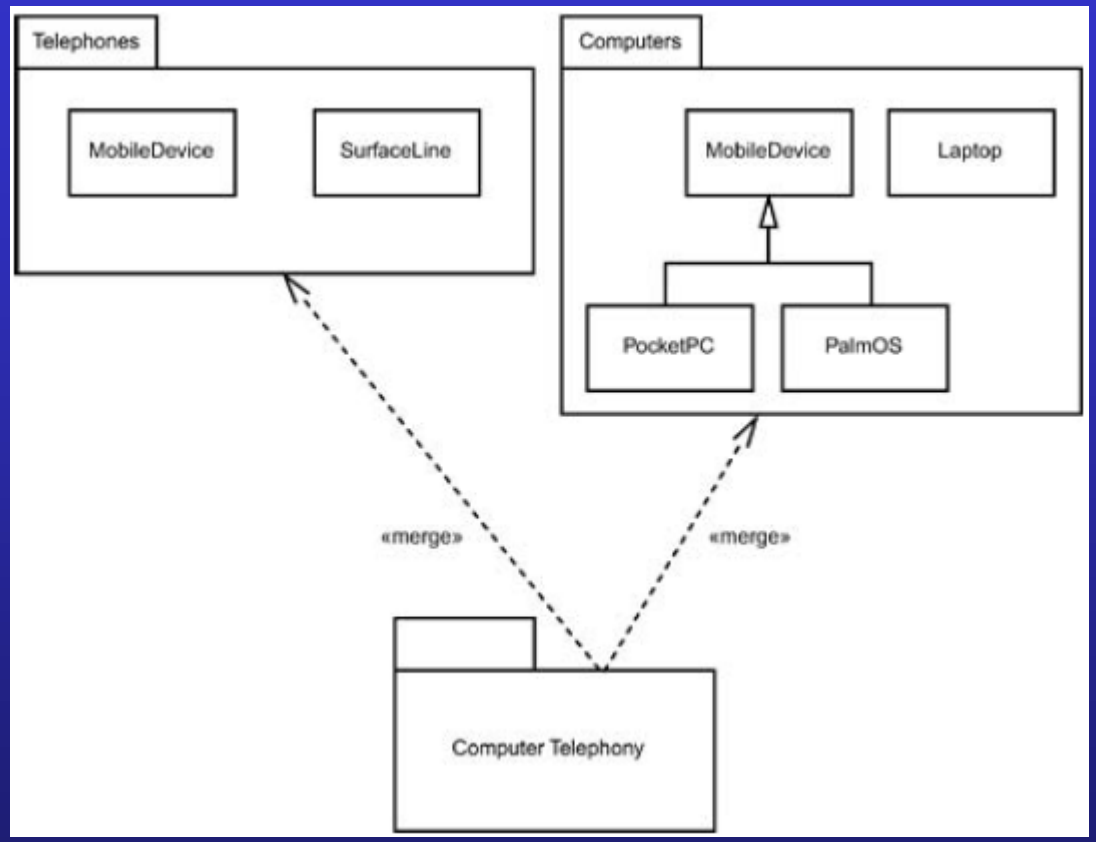
两种连接器

部署图


CreditCardAgency
执行者
Manager
节点
TicketServer
«artifact»
CreditCardCharges.jar
①
«artifact»
ManagerInterface.jar
制品
«artifact»
TicketSeller.jar
«artifact»
TicketDB
TicketServer
1
1
依赖
通信关联
*
*
Kiosk
«artifact»
CustomerInterface.c
SalesTerminal
«artifact»
ClerkInterface.c
«manifest»
节点
«manifest»
②
KioskInterface
ClerkInterface
Customer
Clerk

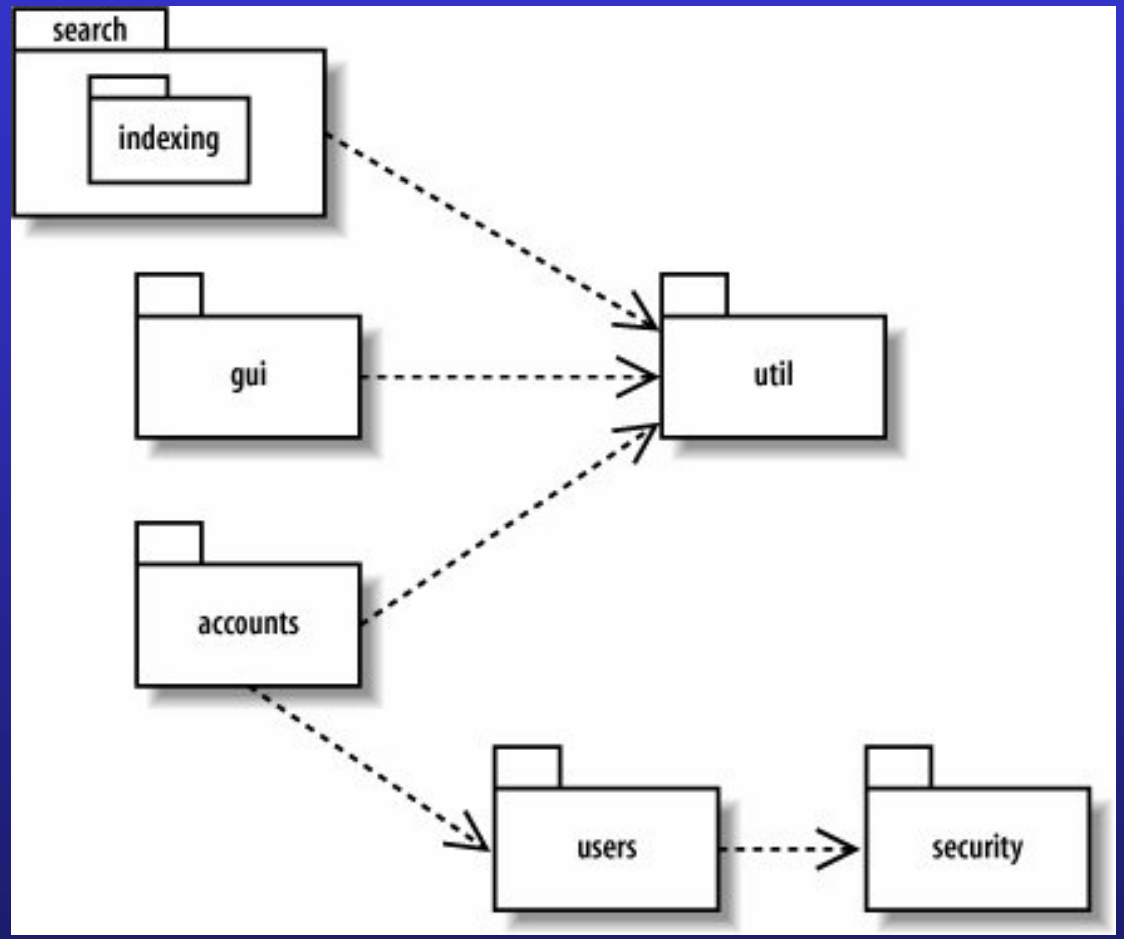
包图





组织各种UML元素

包图



依赖