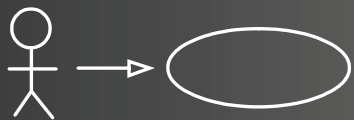


UML全程实作

设计

Think



<http://www.umlchina.com>

核心 workflow

*愿景

*业务建模

选定愿景要改进的业务组织

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

书写用例文档

提升
销售

*分析

类图

序列图

状态图

*设计

建立数据层

精化业务层

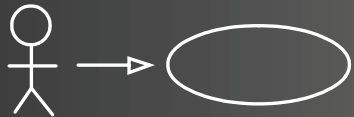
精化表示层

降低
成本



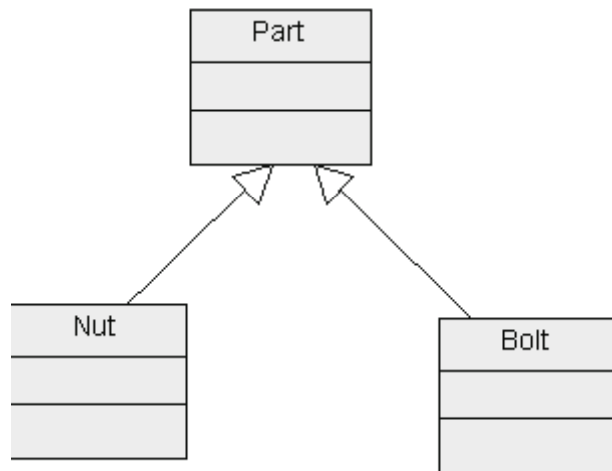
分析和设计

- 分析：提炼核心域知识
- 设计：添加非核心域知识

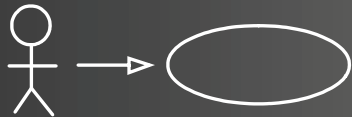


代码映射

——泛化

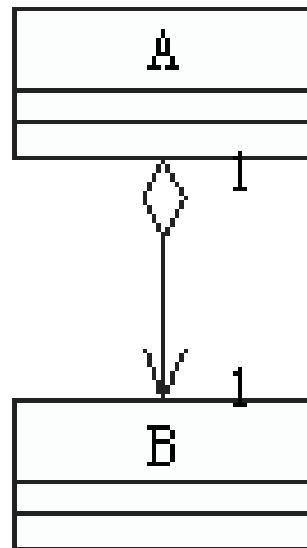


```
public class Part
{
    //...
}
public class Bolt : Part
{
    //...
}
public class Nut : Part
{
    //...
}
```



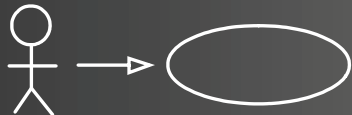
代码映射

——聚合、单向连接（1:1）



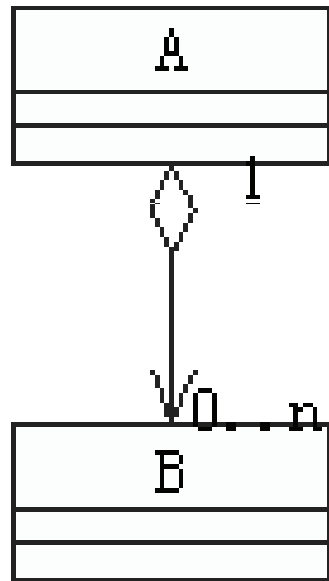
```
public class A
{
    protected B theB;
}

public class B
{
    // ...
}
```



代码映射

——聚合、单向连接（1:n）



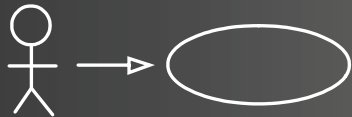
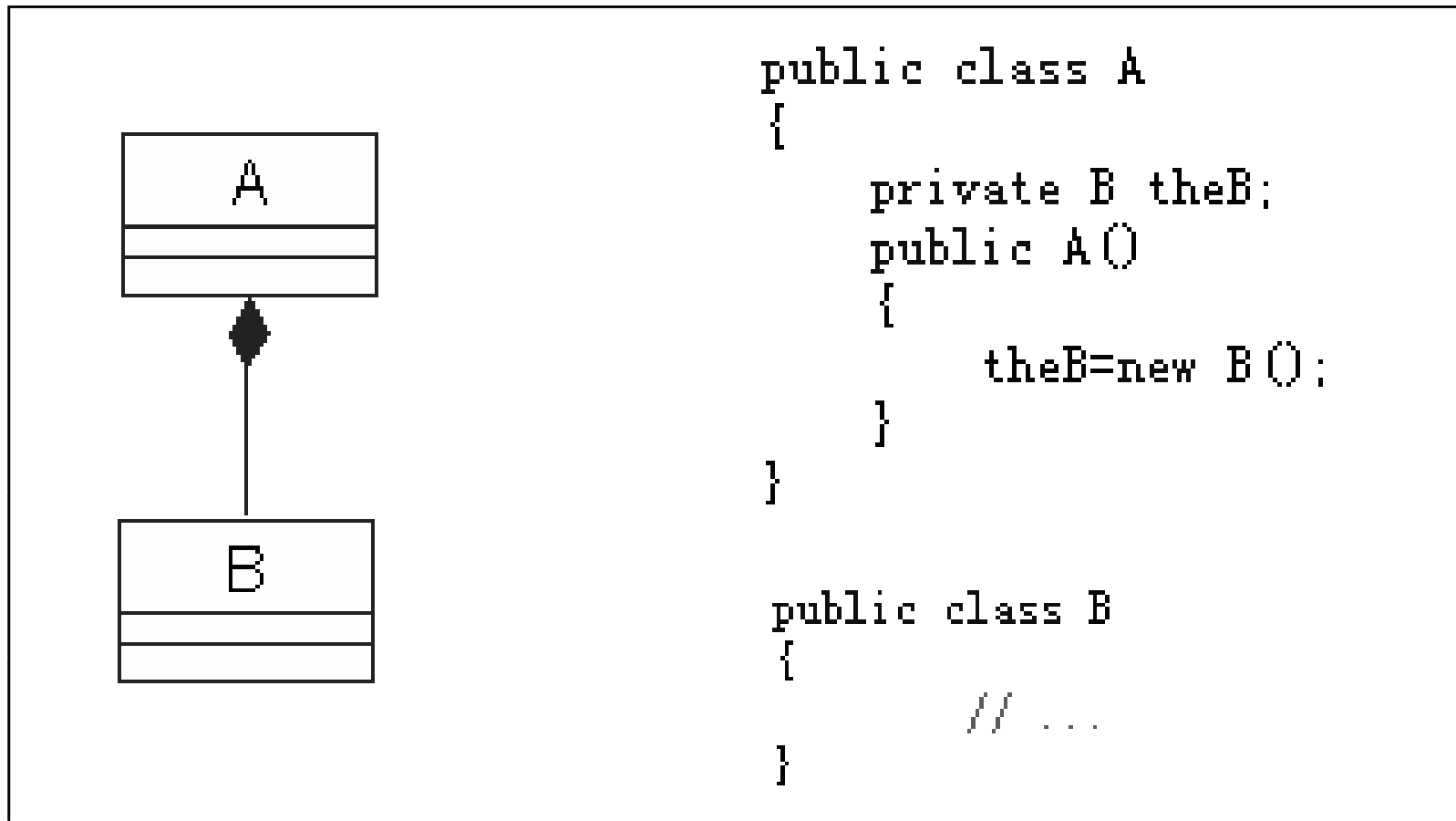
```
public class A
{
    protected ArrayList theBs;
}

public class B
{
    // ...
}
```



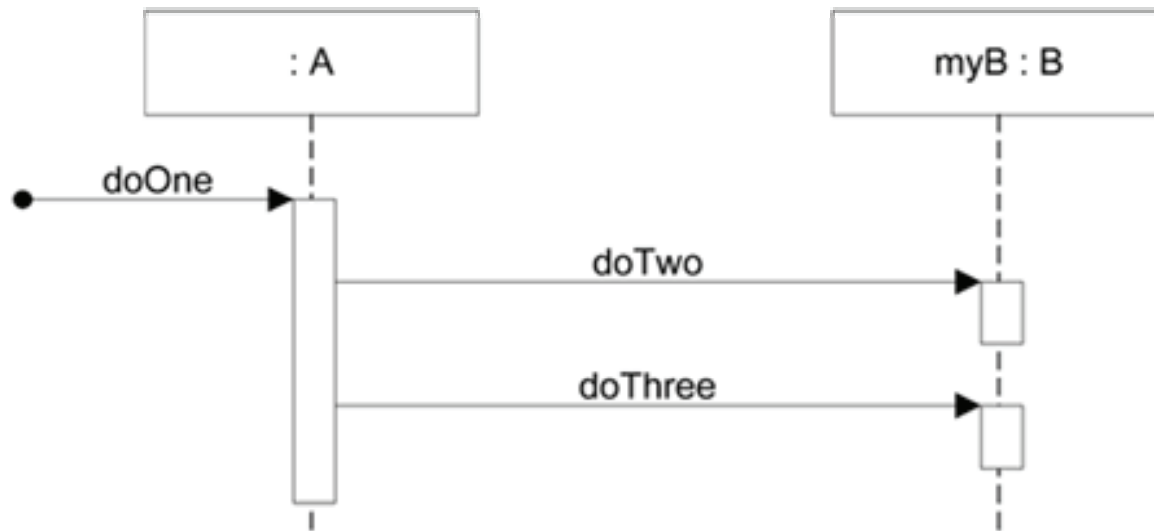
代码映射

——组合



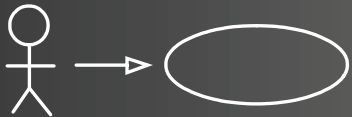
代码映射

——序列图

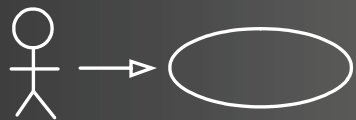
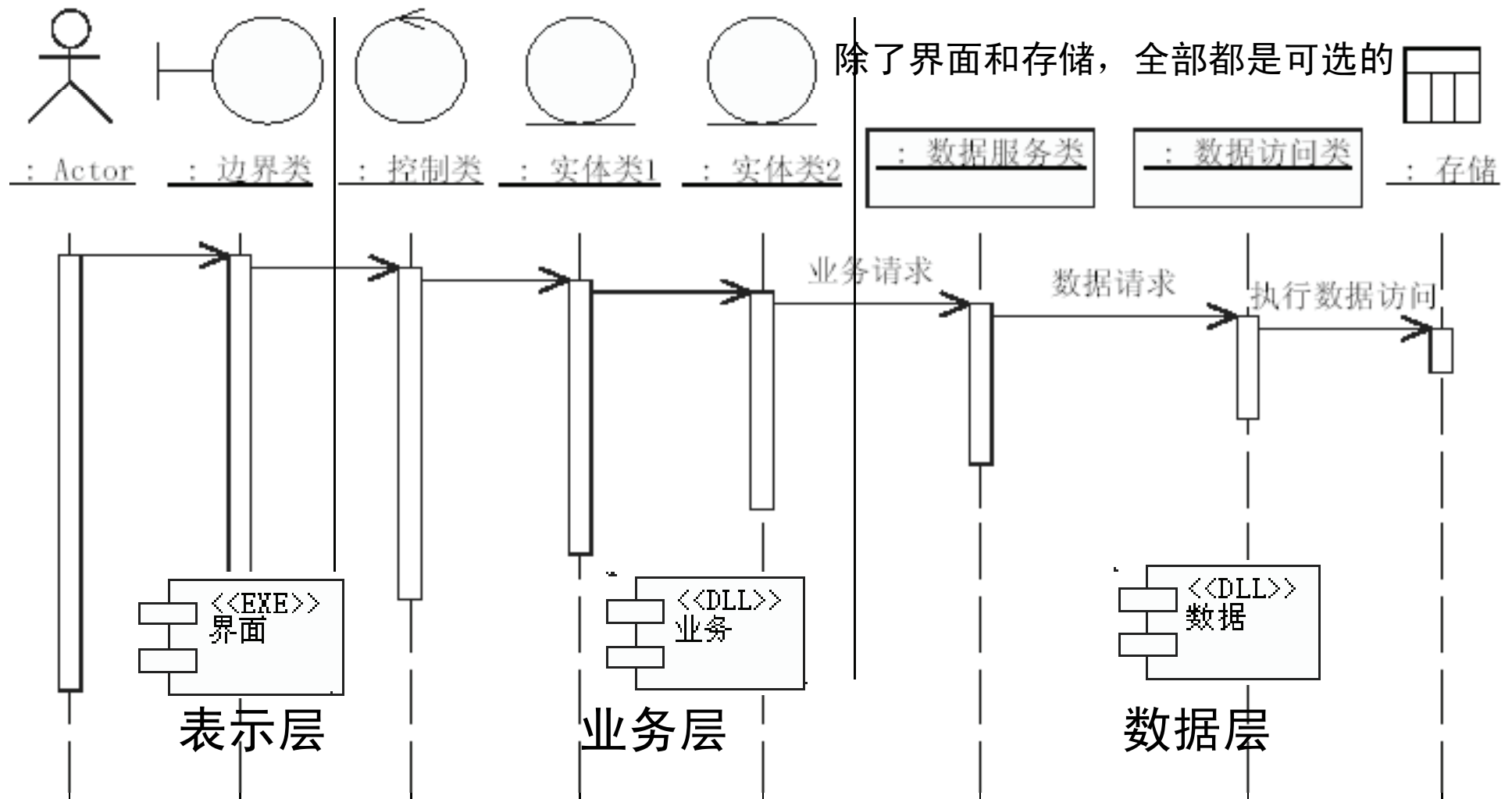


```
public class A
{
    private B myB = new B();

    public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }
    // ...
}
```



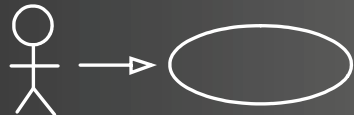
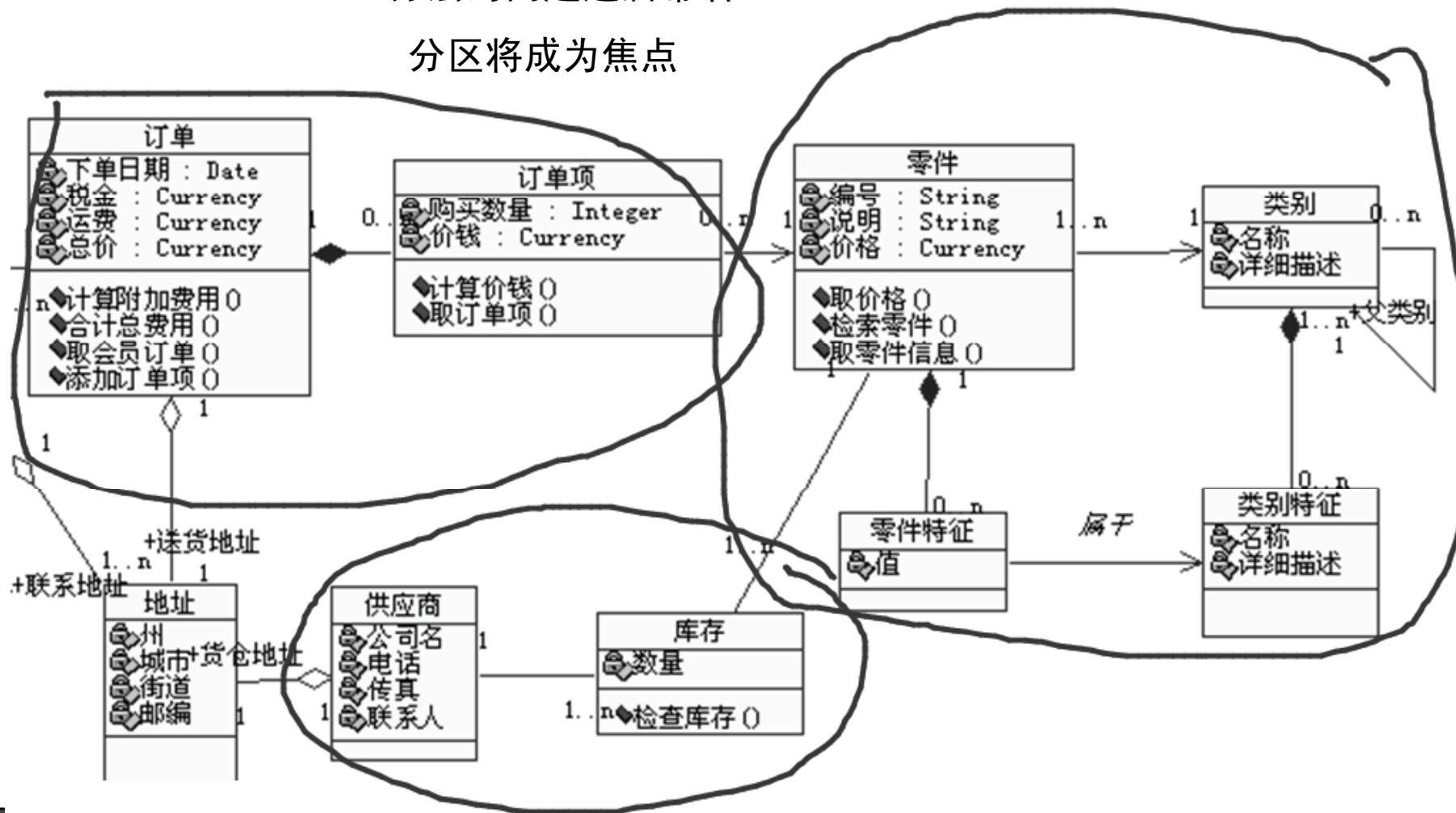
整个软件的层次



分层和分区

一旦分层的问题退居幕后，

分区将成为焦点

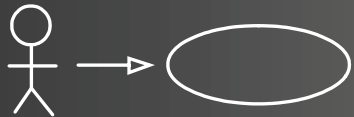


数据层

 映射存储



 构造数据源层



存储：对象持久化

❖ 文件

- ❖ 各种格式（CSV, XML, JSON, YAML…）

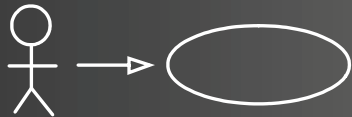
❖ 关系数据库（RDBMS）（最常用）

Microsoft
SQL Server 2005
Enterprise Edition

ORACLE®

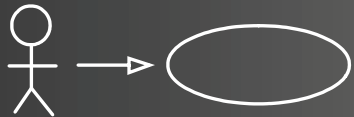
❖ 面向对象数据库（OODBMS）

- ❖  Jasmine（多媒体，大规模集成电路）



关系数据库回顾

这些数据存储结构有什么问题？



关系数据库回顾

❖ 规范化

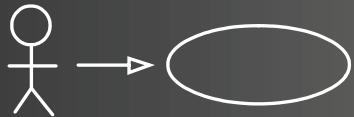
❖ 使得任何数据子集都能够通过基本的SQL操作符获取

❖ 范式

❖ 第一范式：不存在多值字段

❖ 第二范式：非主键字段依赖于主键的整体

❖ 第三范式：非主键字段只依赖于主键

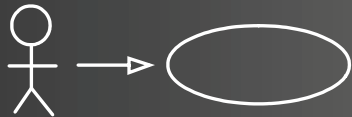


用关系数据库来存储

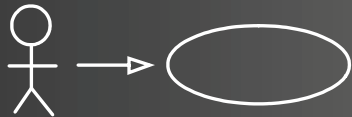
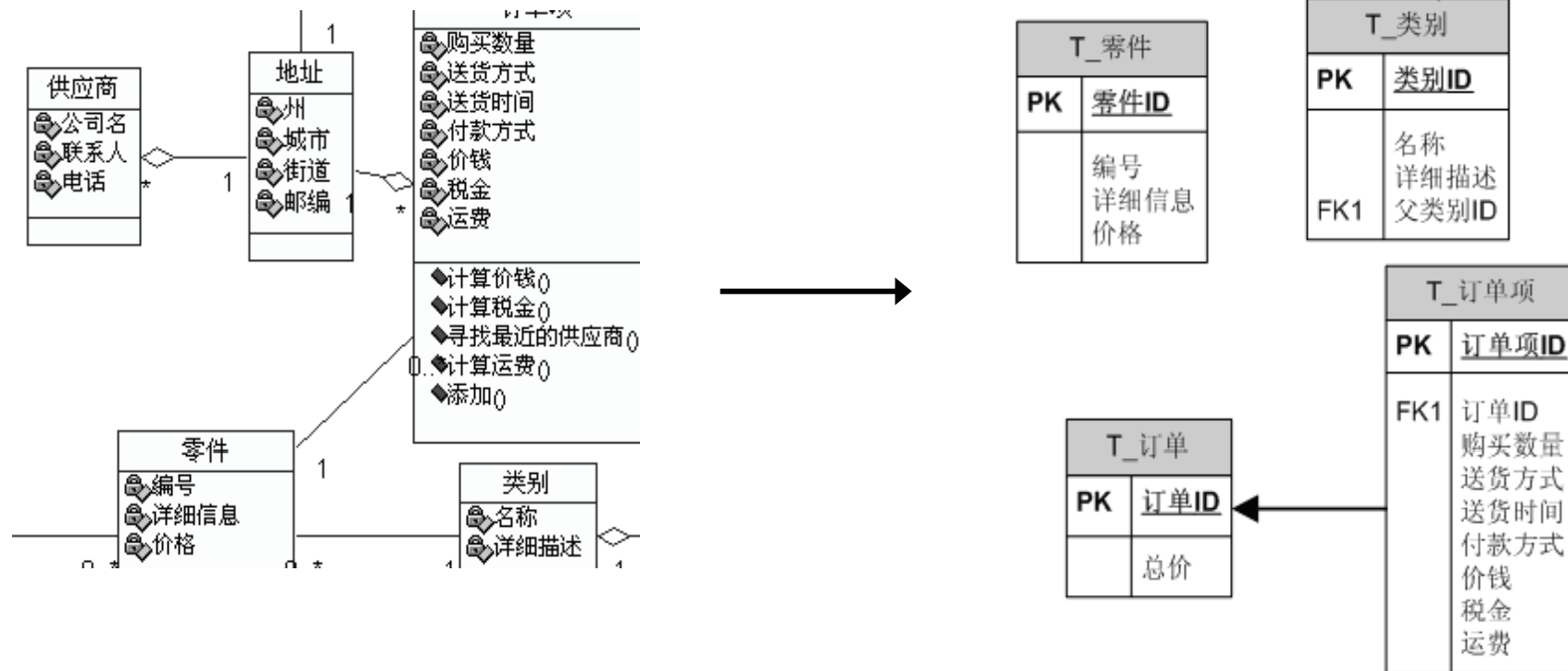
- ❖ 你想把车停在一个面向对象的车库里。把车开进车库，下车，关上车门，然后回到你的房间。当你想出去的时候，只要走进车库，钻进汽车，启动，然后开走。



- ❖ 你想把车停在一个关系数据库的车库里。把车开进车库，下车，卸下车门，将它们放在地上；卸下所有的车轮，将它们放到地上；卸下保险杠及其它的东西。然后回到你的房间。当你想出去的时候，走进车库，先安上车门，再安上保险杠，然后是车轮等等，都安完了，钻进汽车，点火，然后开走。



把实体类映射到关系数据库

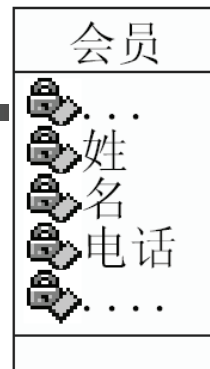


映射类和属性

类



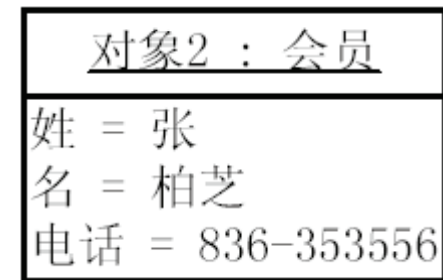
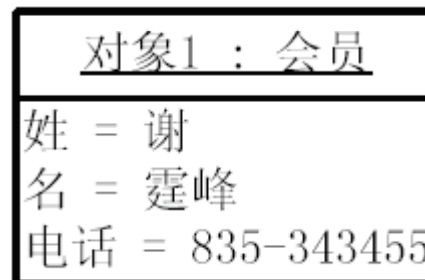
表



对象



行

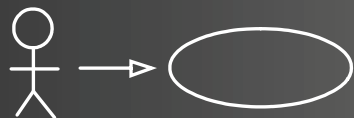


属性

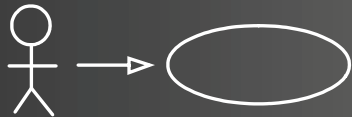
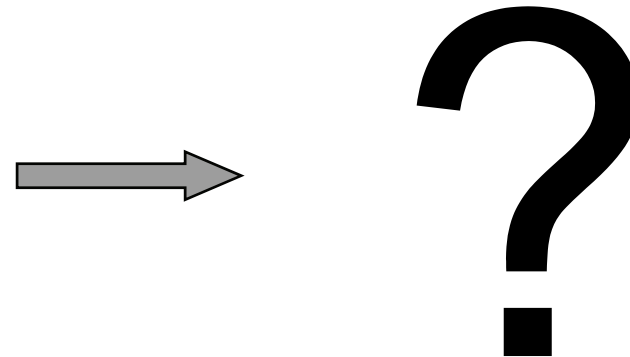
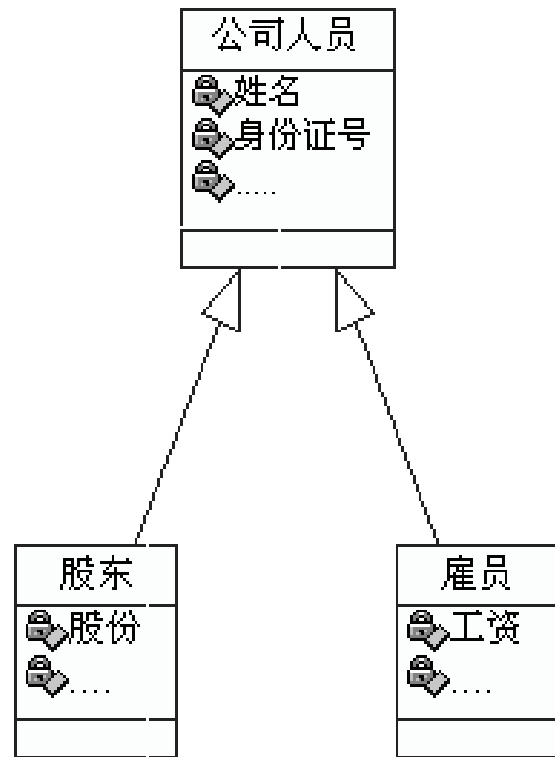


列

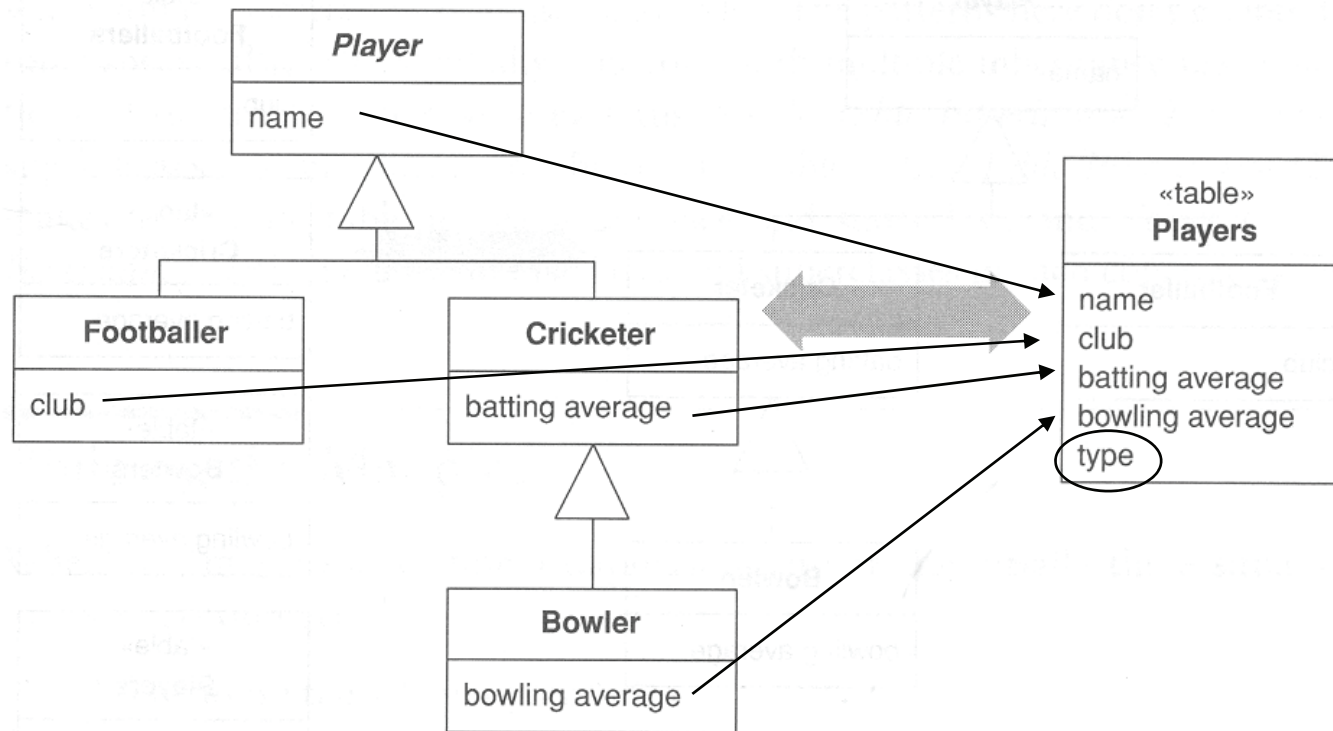
姓	名	电话
谢	霆峰	835-343455
张	柏芝	836-353556



映射泛化关系



映射泛化关系



❖ 优点

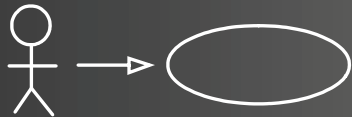
- ❖ 易于修改
- ❖ 避免联接

❖ 缺点

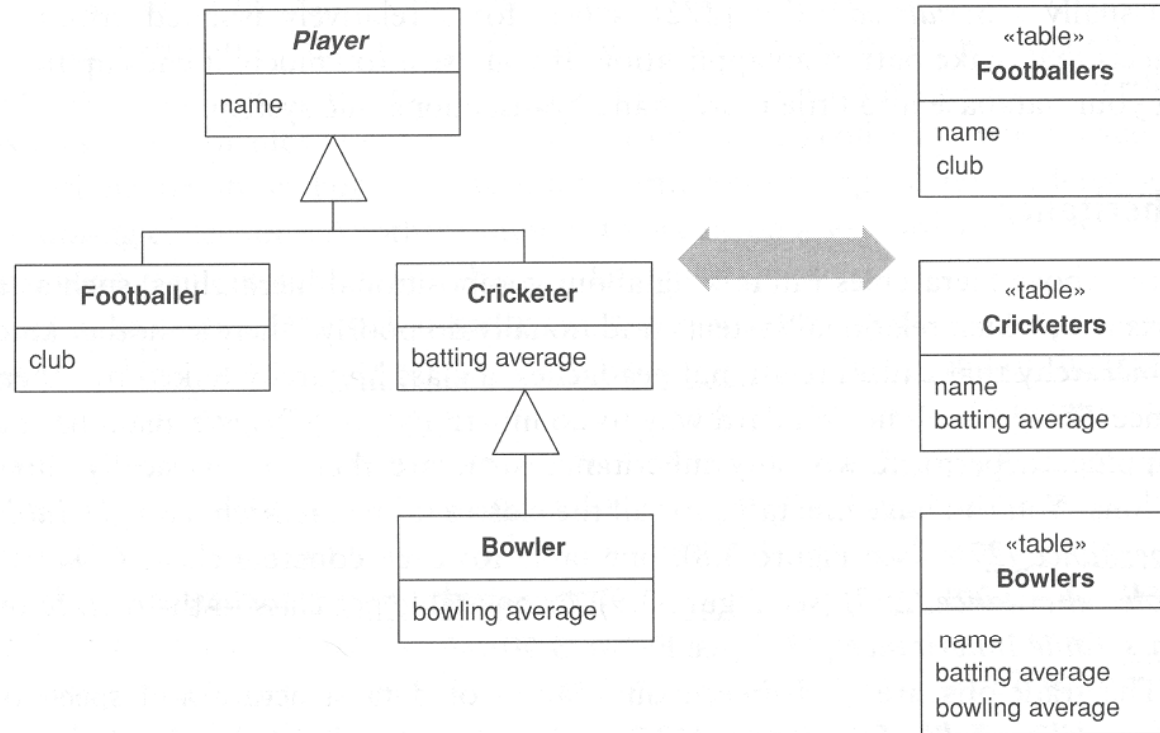
- ❖ 浪费空间

Figure 3.8 Single Table Inheritance (278) uses one table to store all the classes in a hierarchy.

单表继承—整个类族一个表



映射泛化关系



❖ 优点

❖ 避免联接

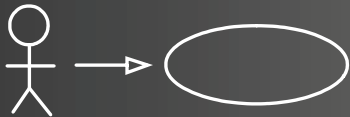
❖ 缺点

❖ 应变差（超类改变、层次结构改变...）

❖ 引用完整性有问题

Figure 3.9 Concrete Table Inheritance (293) uses one table to store each concrete class in a hierarchy.

具体表继承—可以实例化的类有表



映射泛化关系

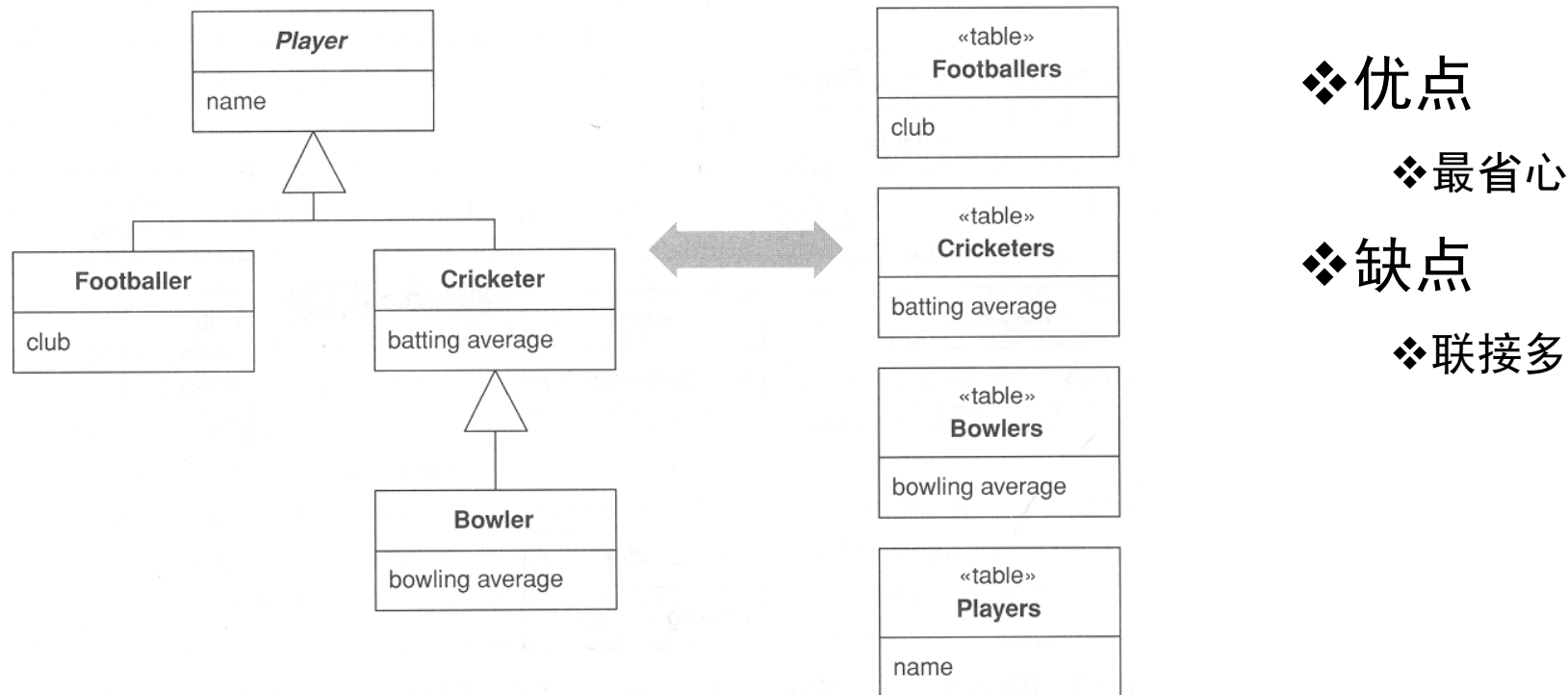
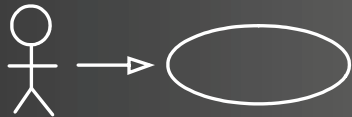
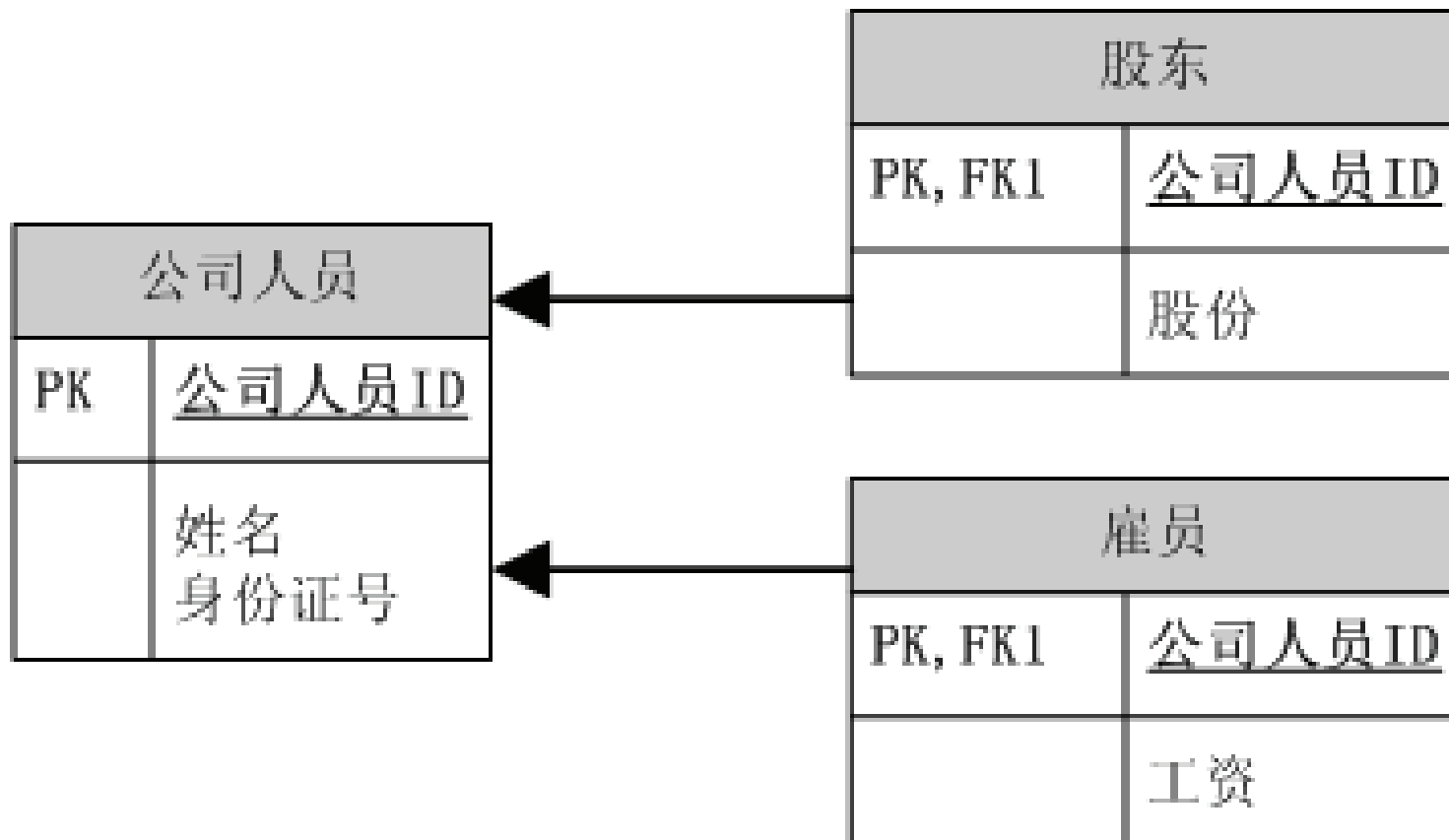


Figure 3.10 Class Table Inheritance (285) uses one table for each class in a hierarchy.

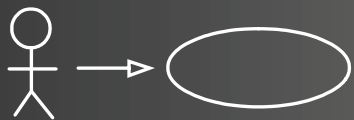
类表继承—每个类一个表



映射泛化关系

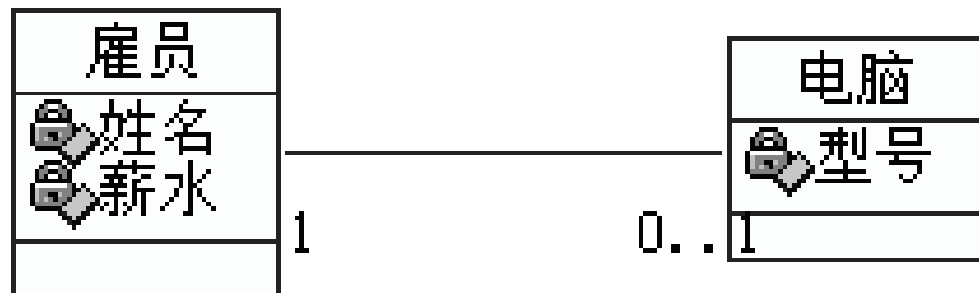


超类子类都映射成表，超类主键作为所有类的主键

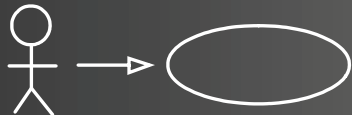
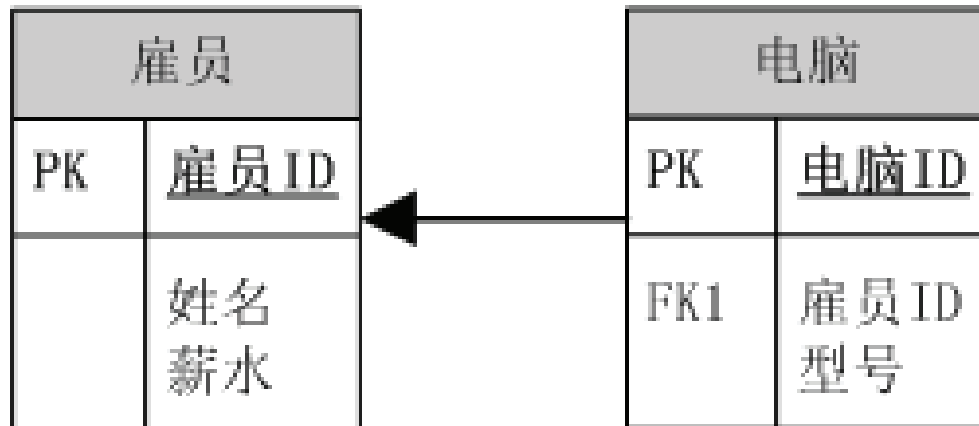


映射连接关系（1）

——1对0..1

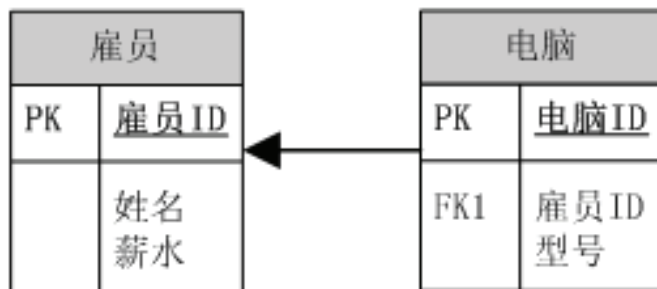
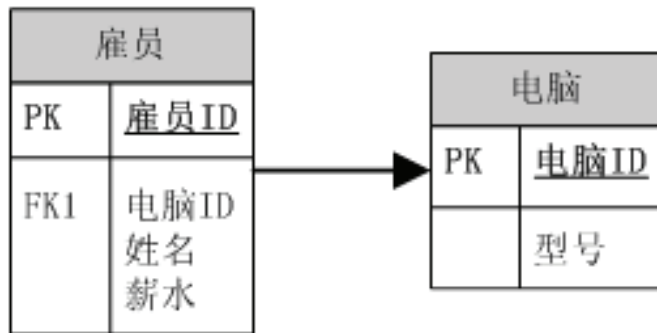


外键放在0..1一端

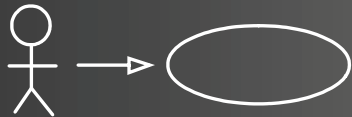


映射连接关系（2）

——1对1

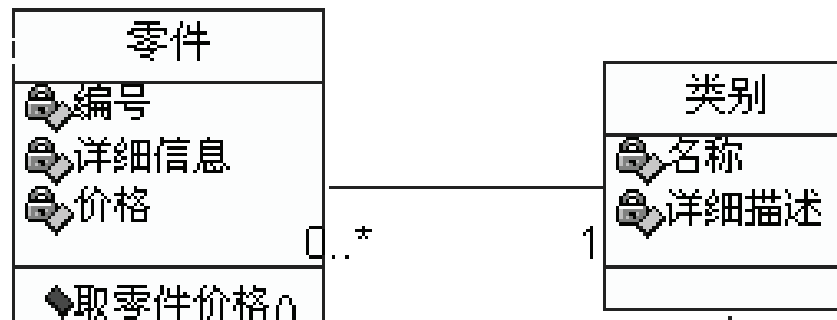


外键放在任意一端

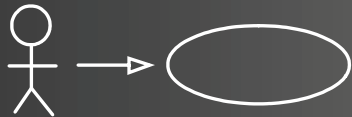
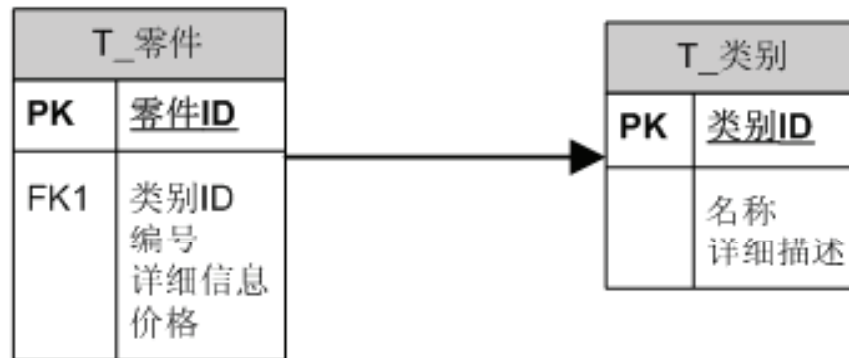


映射连接关系（3）

——1对多

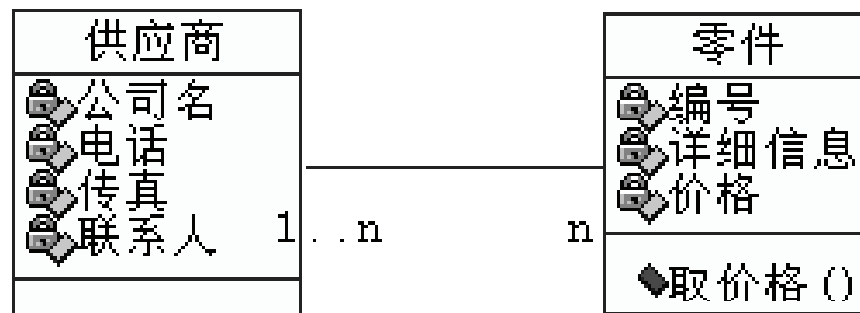


外键放在“多”方

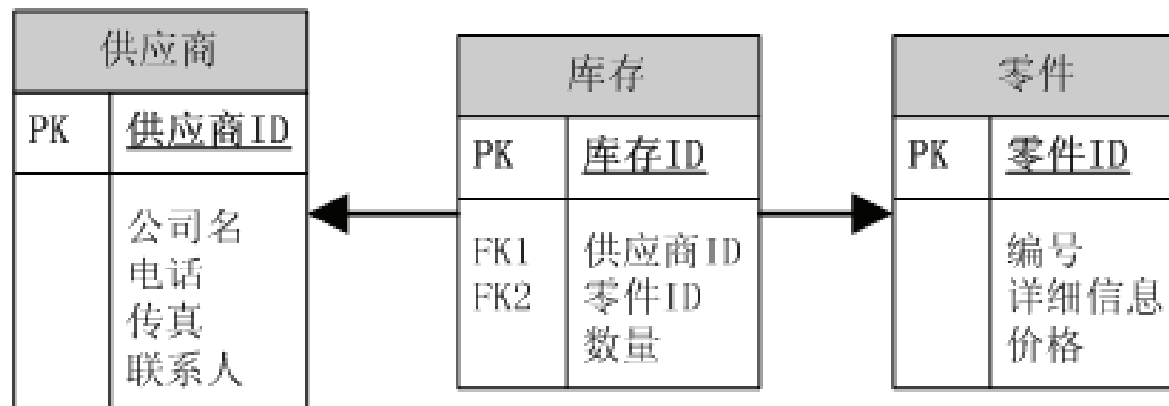


映射连接关系（4）

——多对多

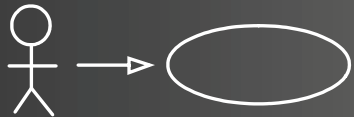


添加第三个表

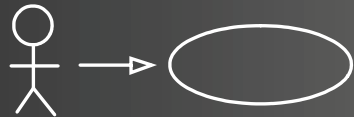
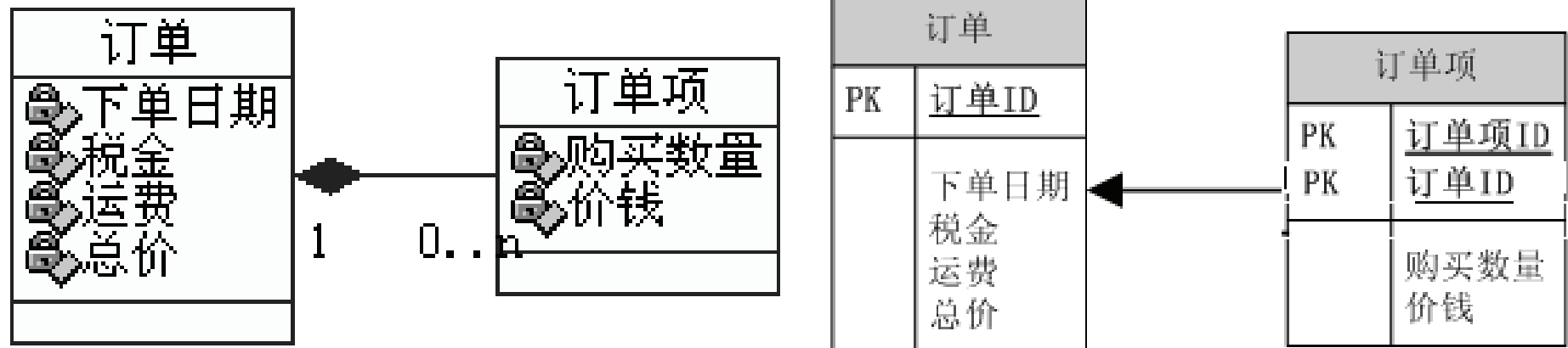


映射聚合关系

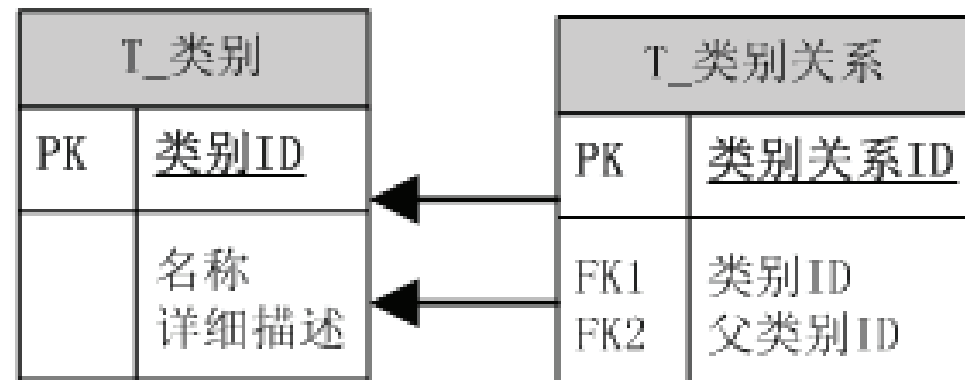
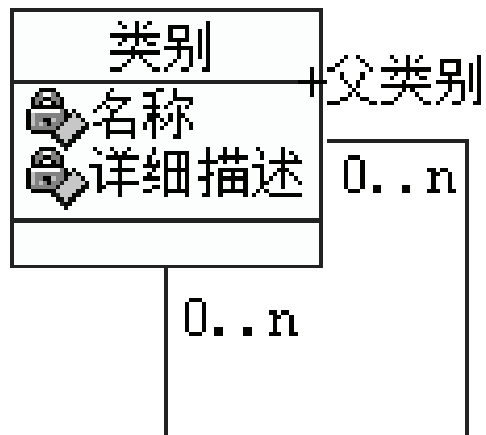
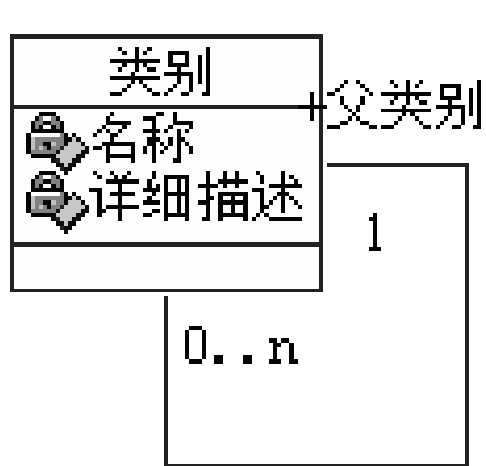
- 映射规则同连接



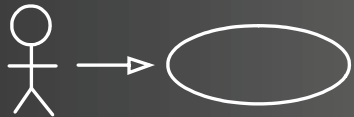
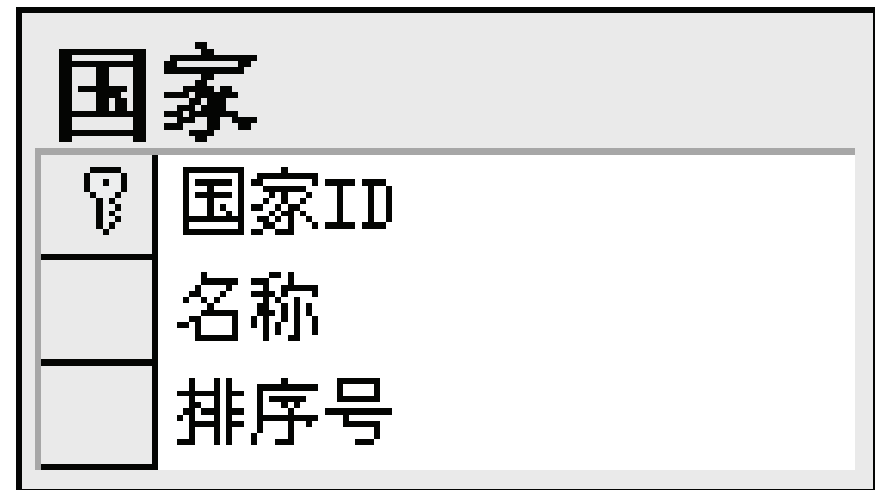
映射组合关系



映射自反关联

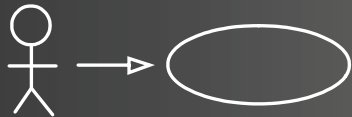


值列表的引入



主键的选择

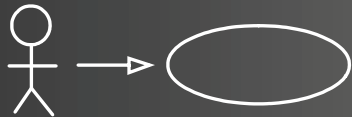
- ❖ 有意义主键还是无意义主键
- ❖ 主键的类型
- ❖ 获得主键的手段



讨论：主键的选择

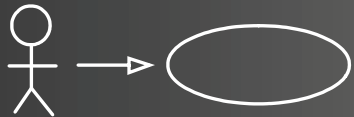
❖ 一个企业组织，“职员”应该用什么作为主键？

- 姓名
- 工号（03012045）
- 身份证号（340205740801203）
- 系统添加的ID



主键的作用

- ❖ 唯一标识记录
- ❖ 被其他表引用为外键
- ❖ 要求：唯一、恒定



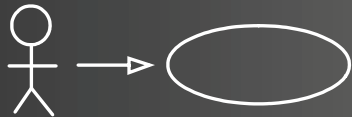
主键不应有业务含义

- ❖ 有业务含义，意味着可能潜伏着变化
- ❖ 任何对主键的修改都可能导致巨大的工作量
- ❖ 输入错误影响
- ❖ 传输安全问题



代理主键的好处

- ❖ 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

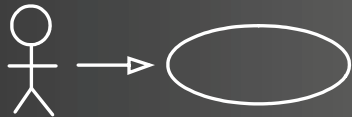


代理主键——注意！

❖ 隐藏代理主键

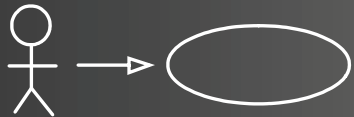
- ❖ 不要让用户在屏幕或报表上看见

- ❖ 不要让用户输入



主键的类型

- ❖ 长整：常用
- ❖ 字符串：相等性检查稍慢
- ❖ 时间日期：可能带有意义，移植问题



主键的获得

❖ 数据库自动生成

- ❖ 自动生成域

- ❖ 数据库计数器：没有统一标准

❖ GUID

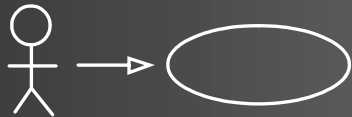
- ❖ 保证唯一：网卡、芯片、纳秒时间

- ❖ 键太长：性能问题、索引

❖ 自己产生

- ❖ Max：糟糕

- ❖ 键表：（名字、下一个值）



为性能而微调

❖ 非规范化（微调性能，慎用）

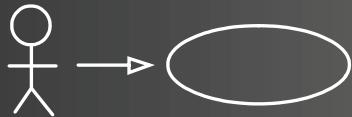
- ❖ 冗余列——方便检索（Total, uppercase…）

- ❖ 冗余表（远端外键）

❖ 原则

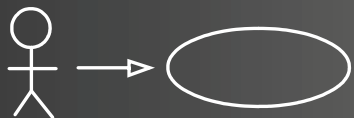
- ❖ 先追求局面上的大赢

- ❖ 出现不可调和的性能问题时，再作微调



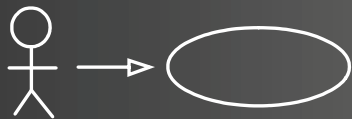
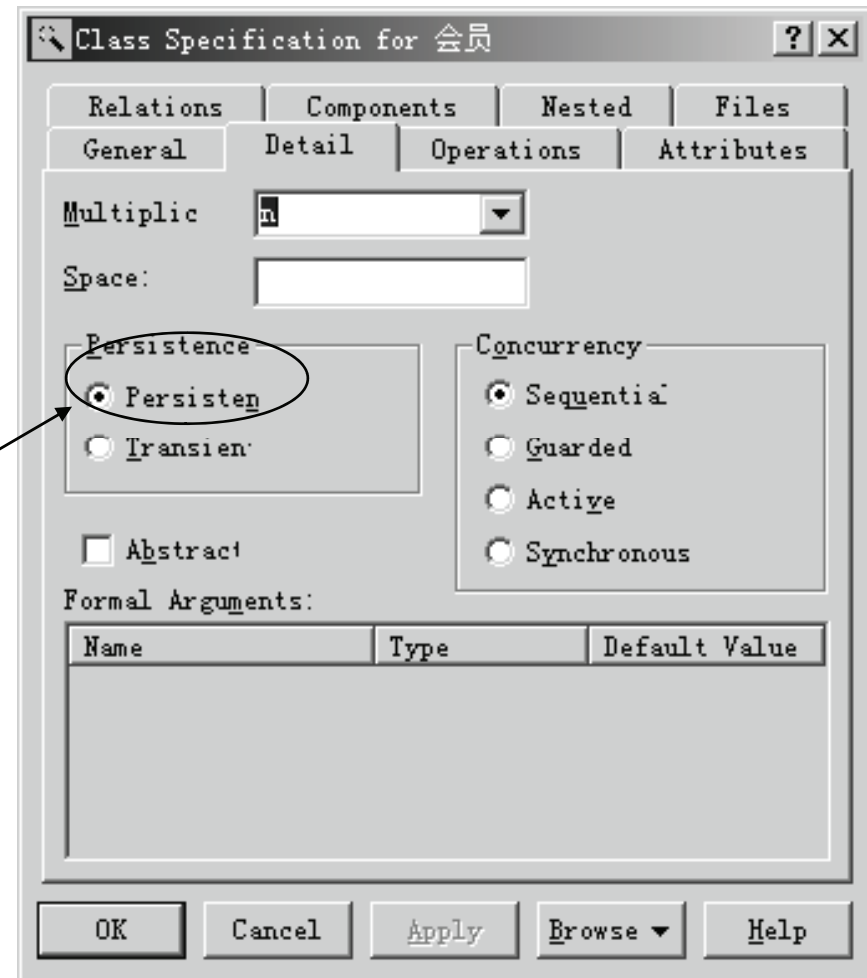
映射数据库

——练习

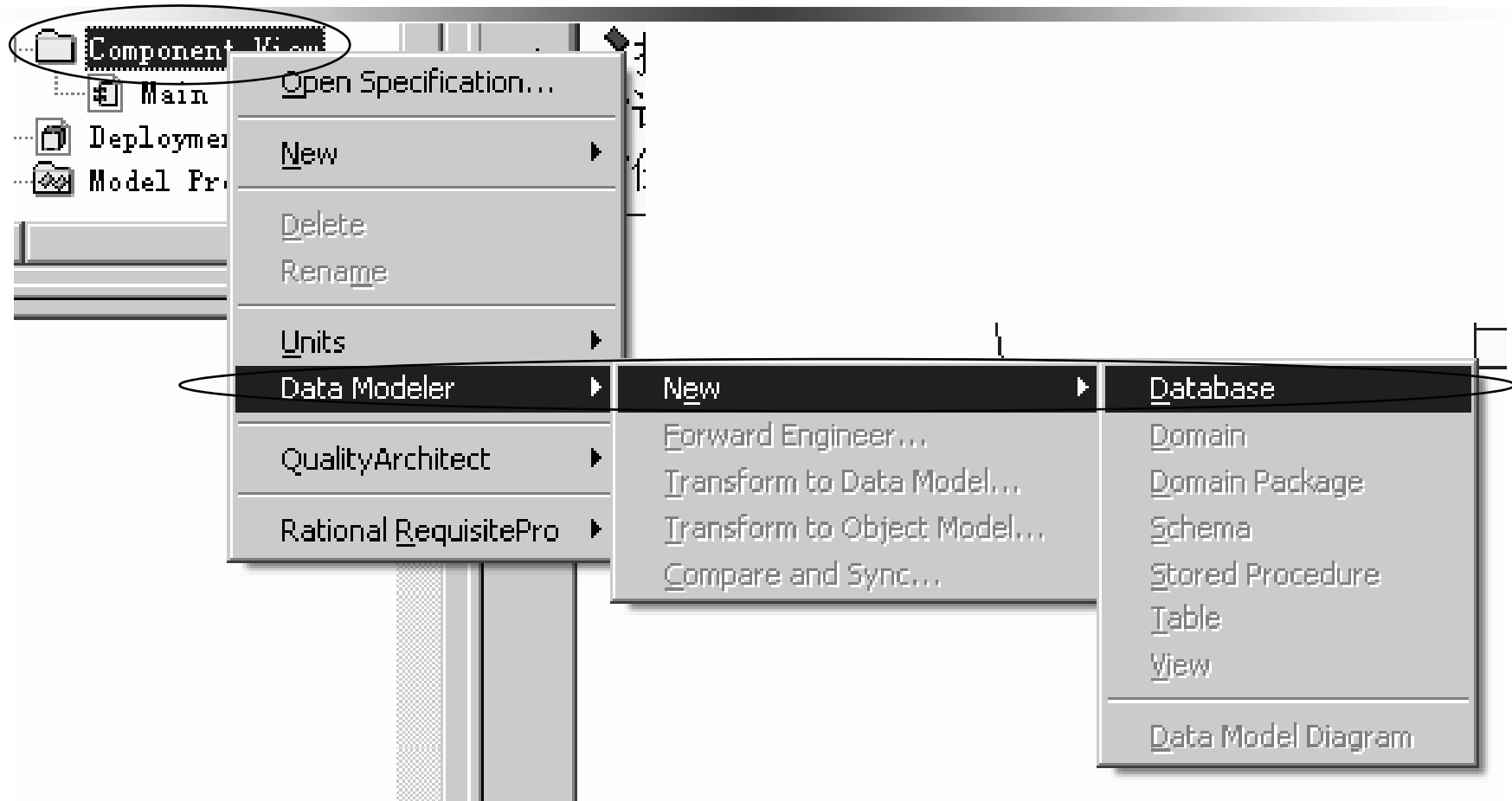


建模工具自动映射（1）

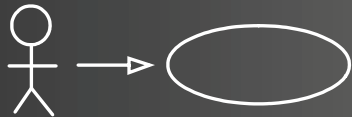
需要持久存储的类要
设置成Persistent



建模工具自动映射（2）

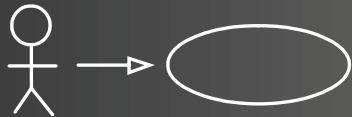
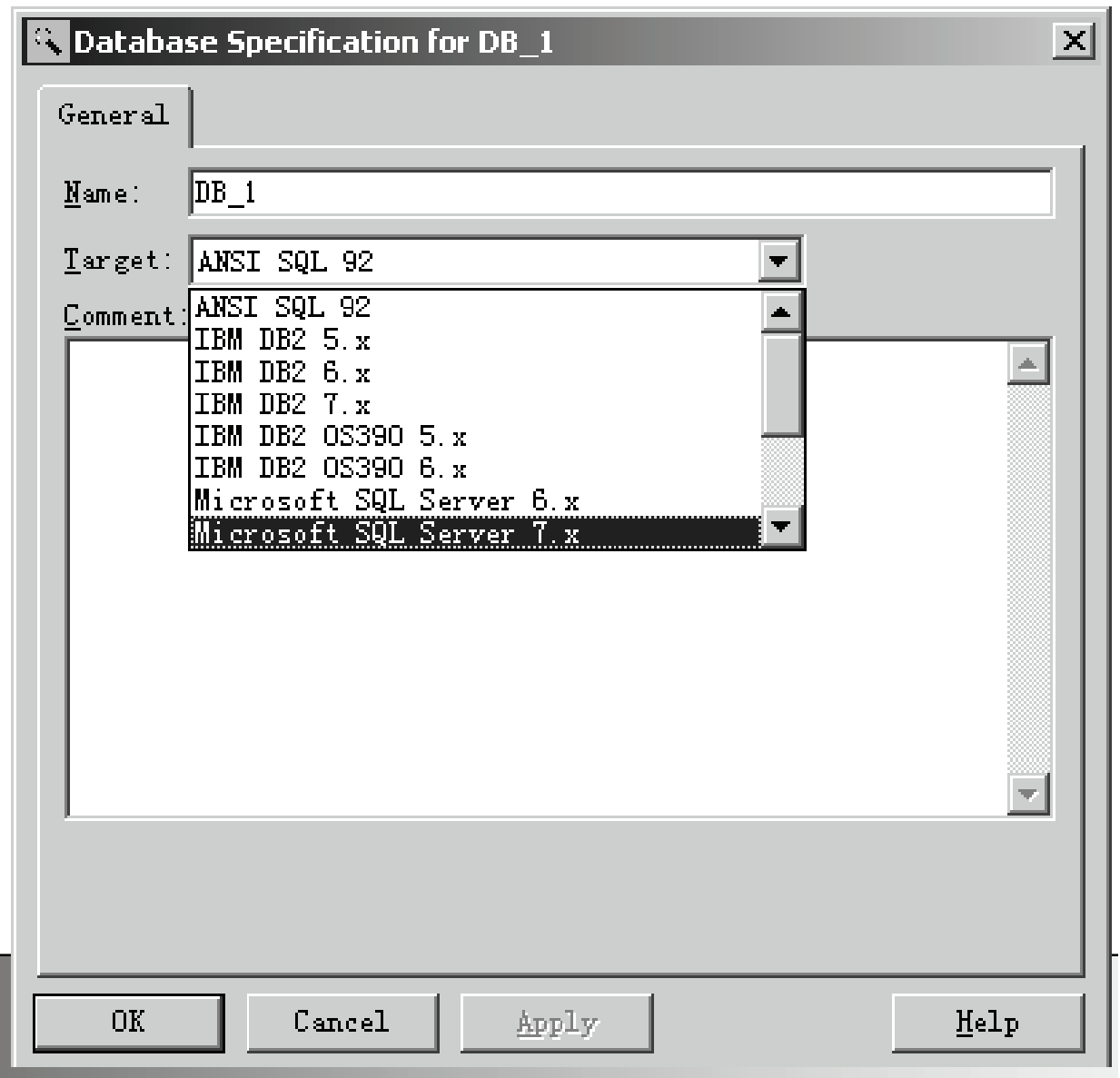


建立数据库“构件”

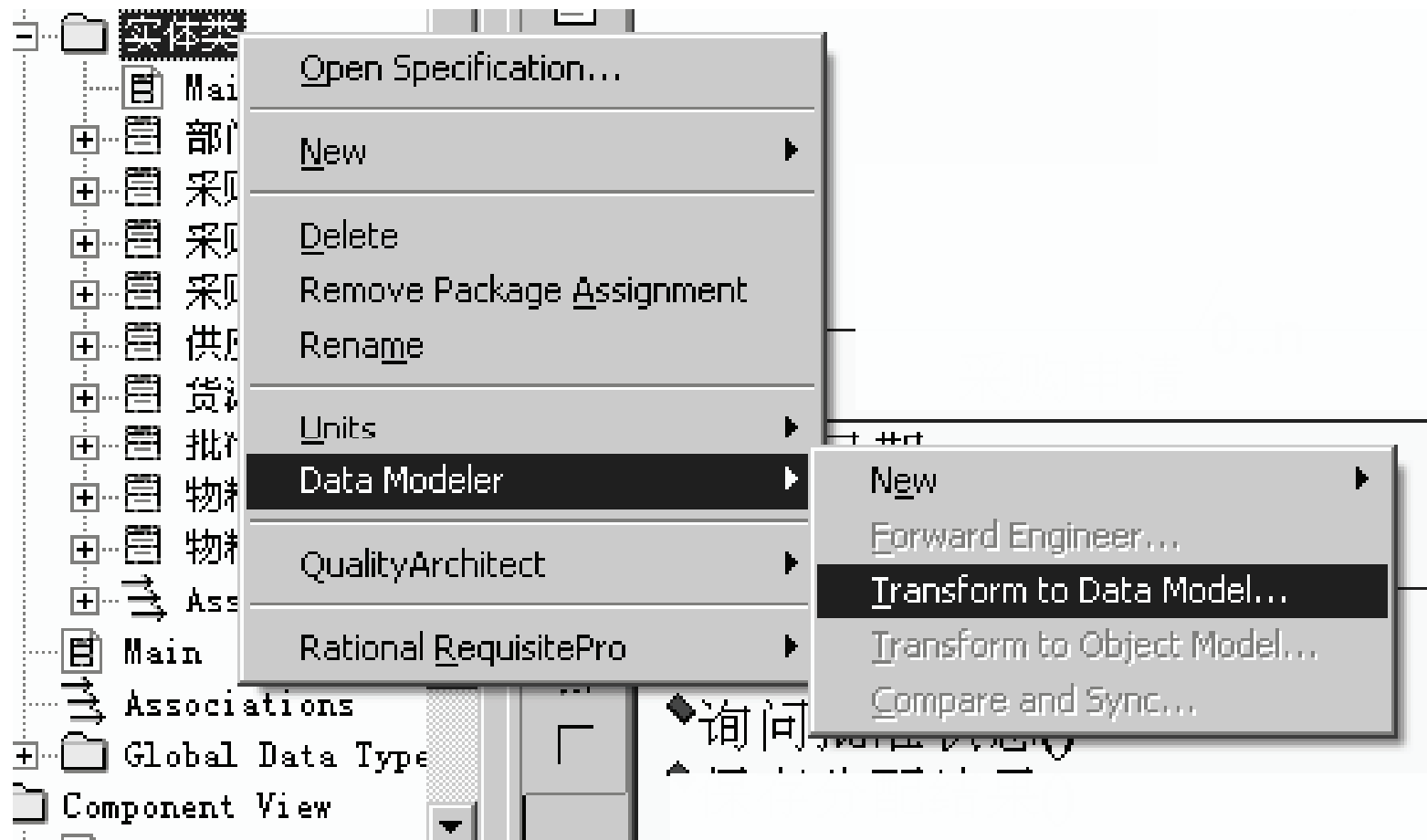


建模工具自动映射（3）

选择数据库平台



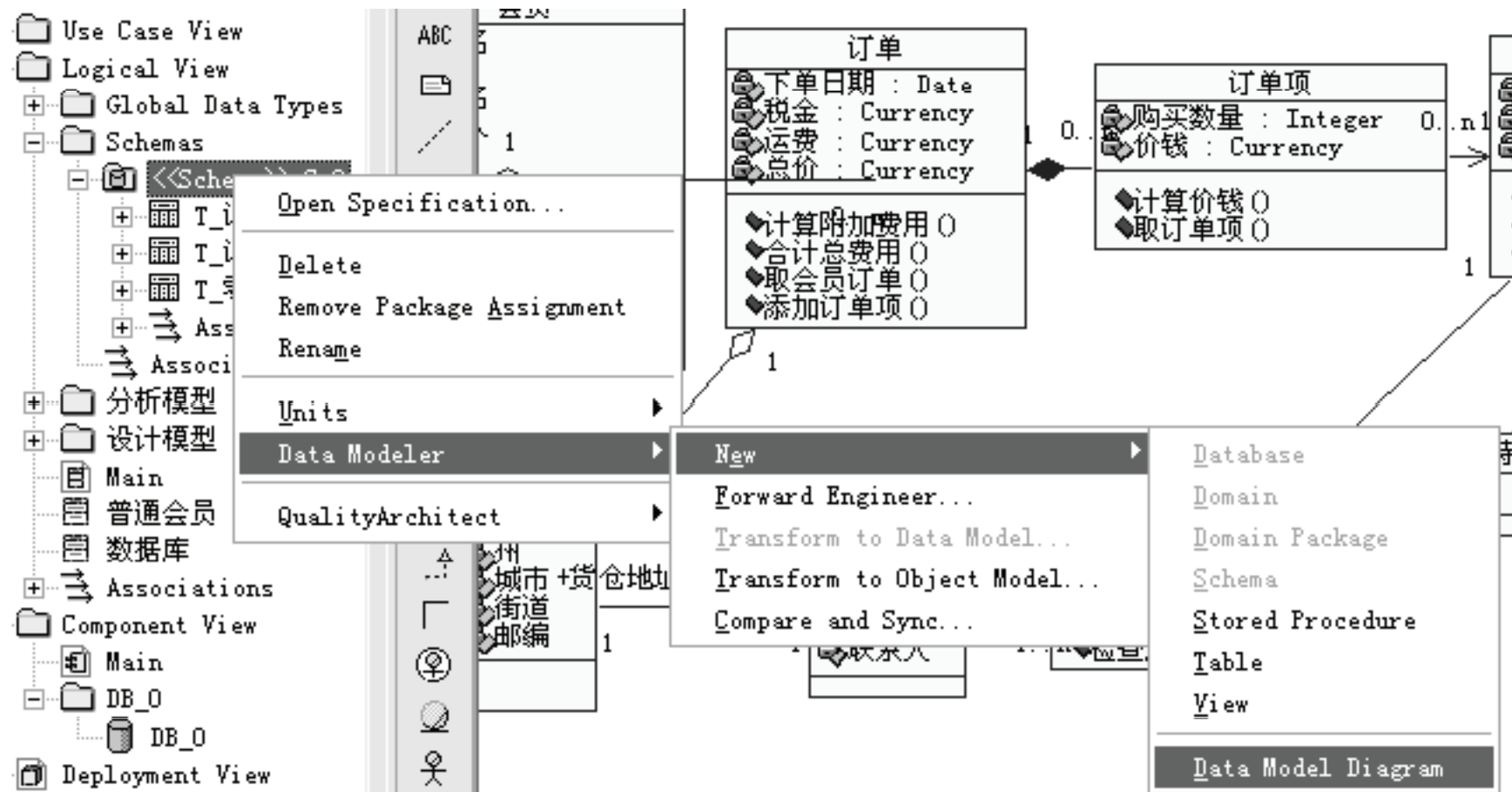
建模工具自动映射（4）



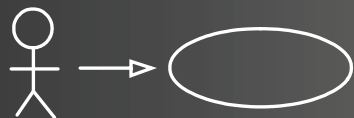
建立Schema



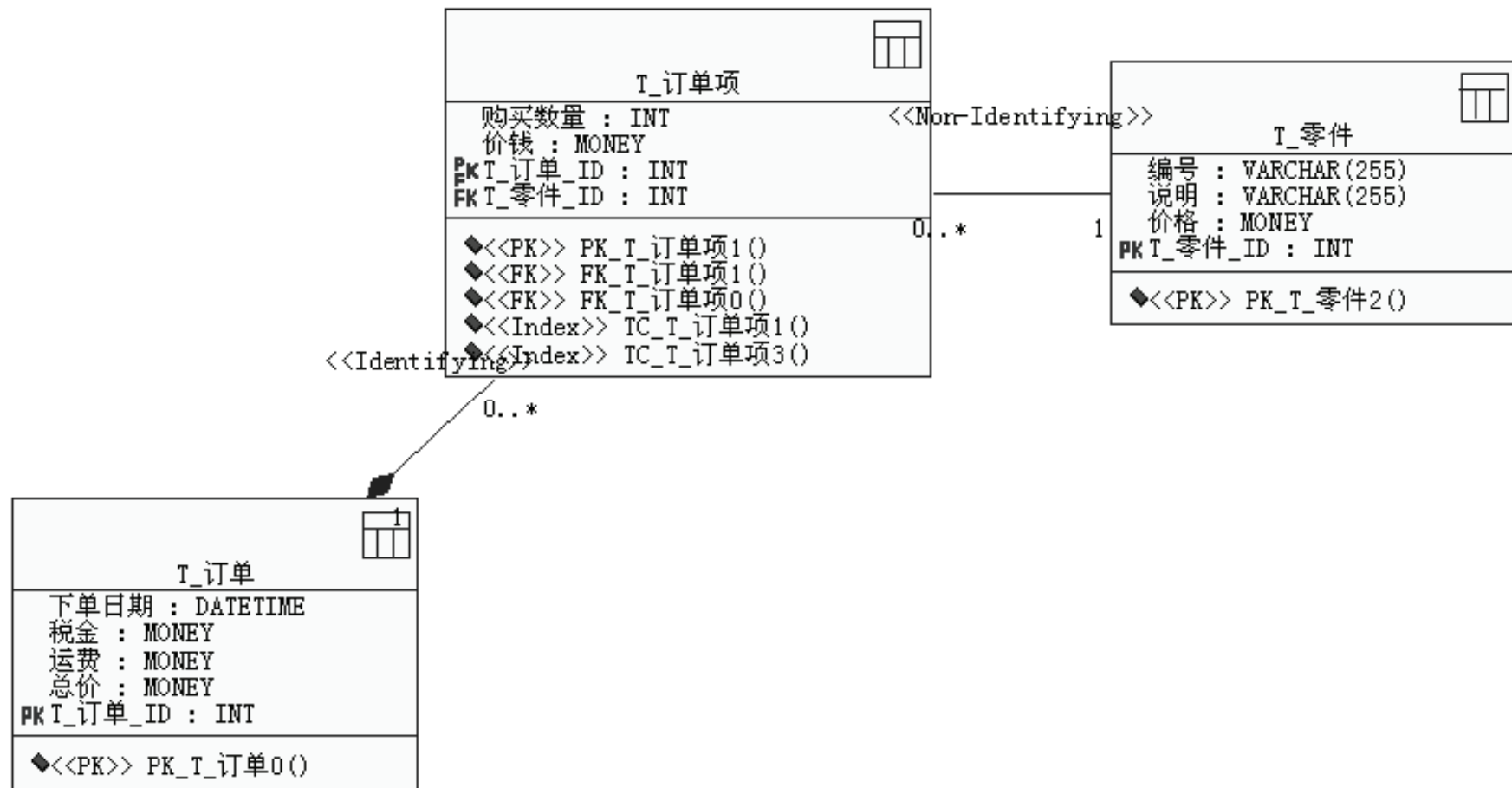
建模工具自动映射 (5)



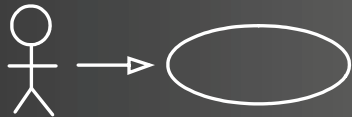
建立数据模型图



建模工具自动映射（6）

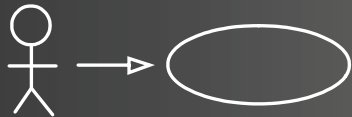
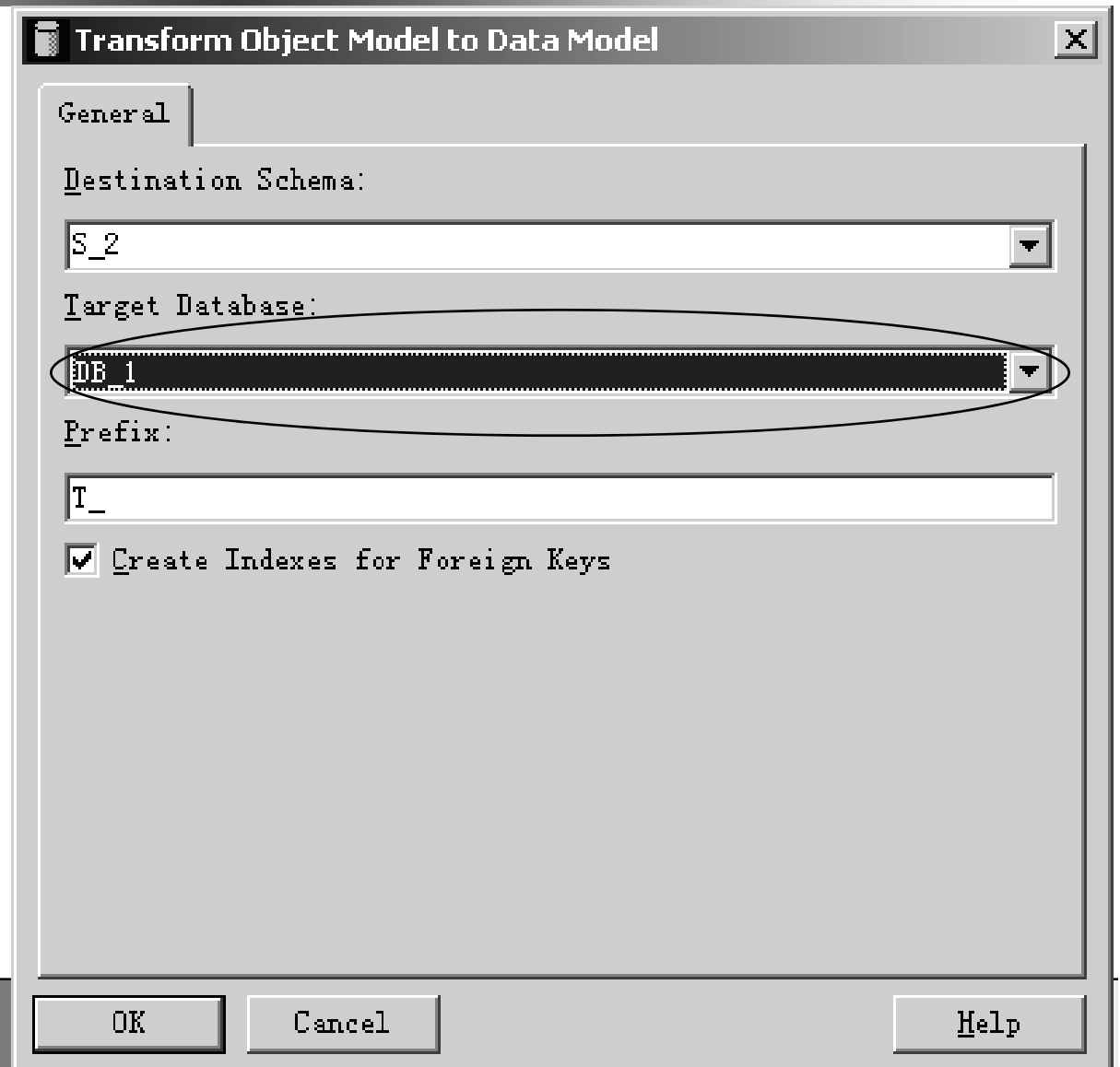


在数据模型图中查看Schema

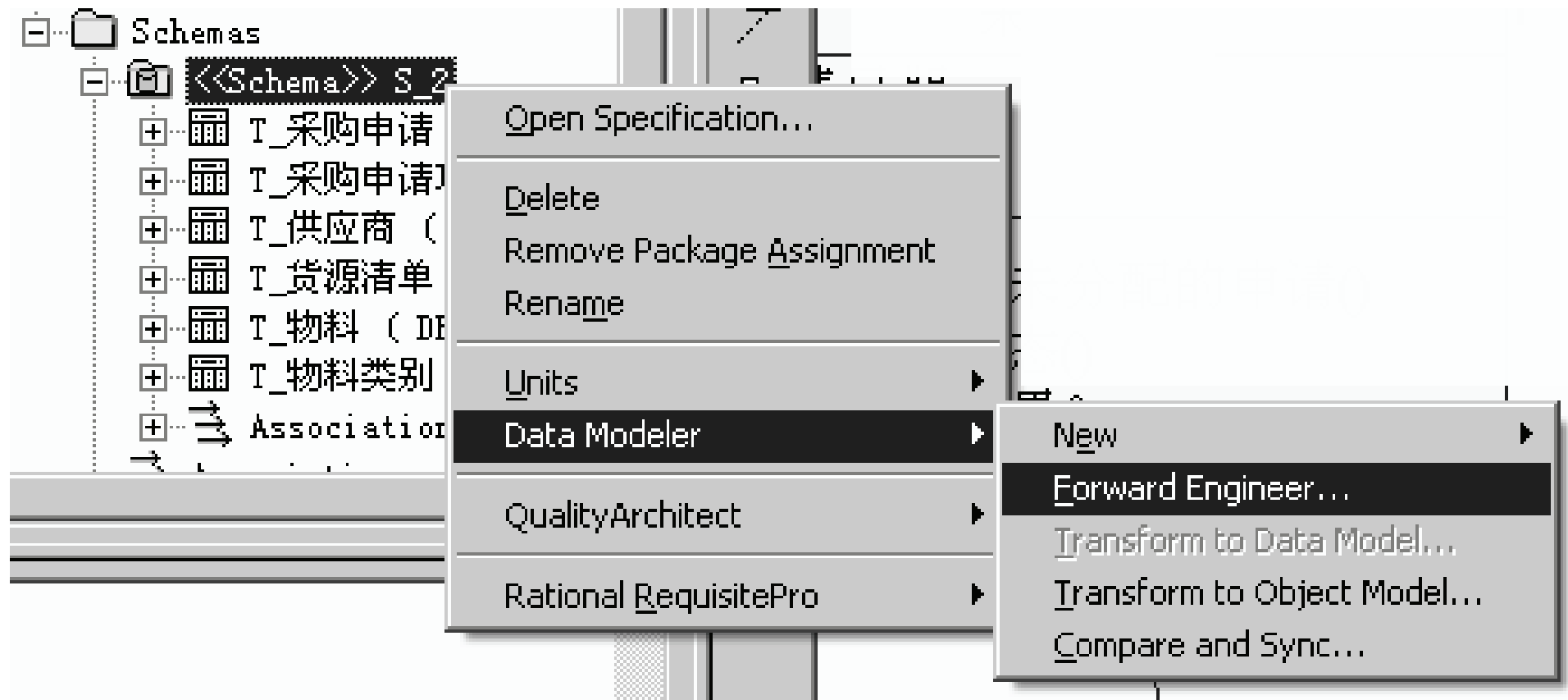


建模工具自动映射（7）

把Schema指向数据库



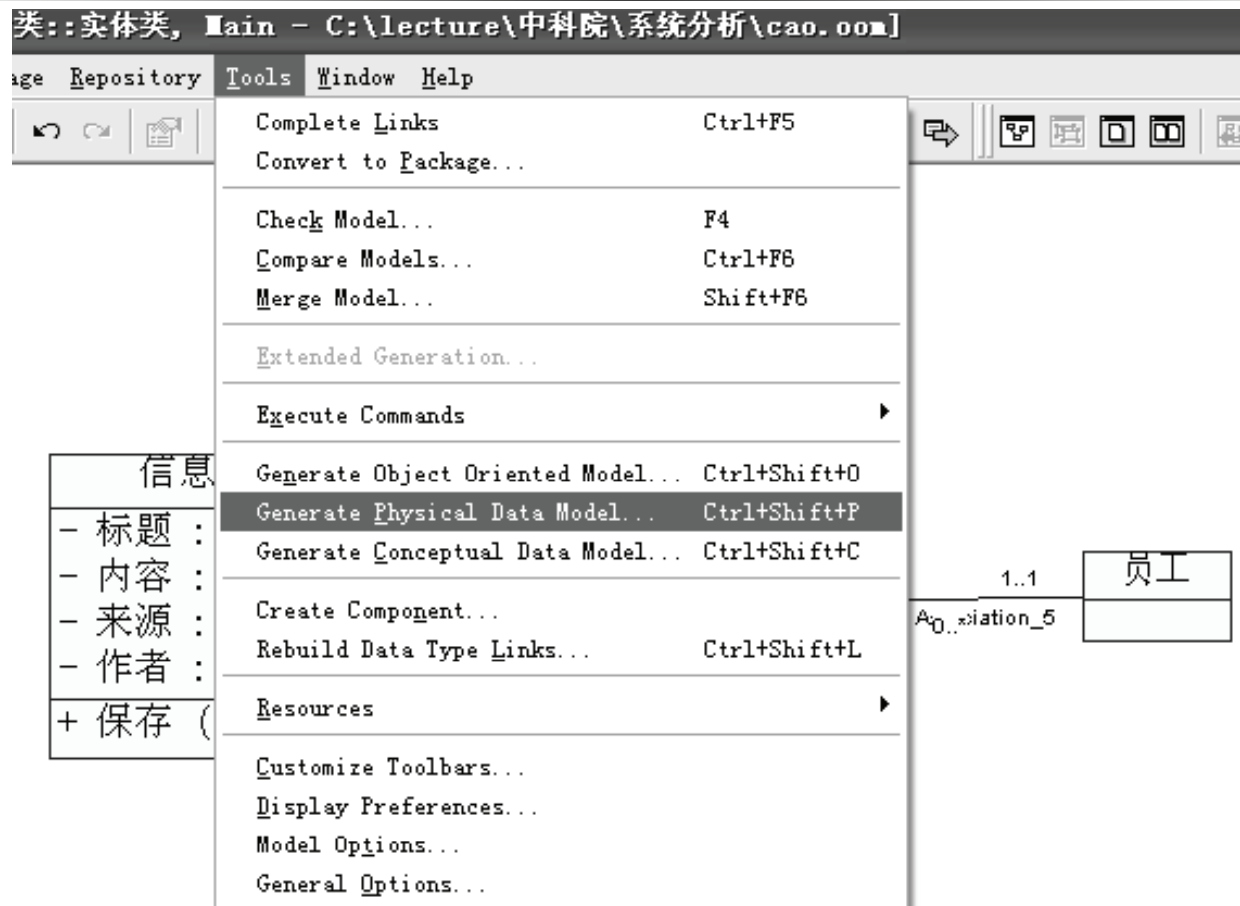
建模工具自动映射（8）



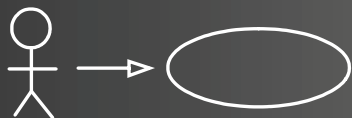
转换！



建模工具自动映射

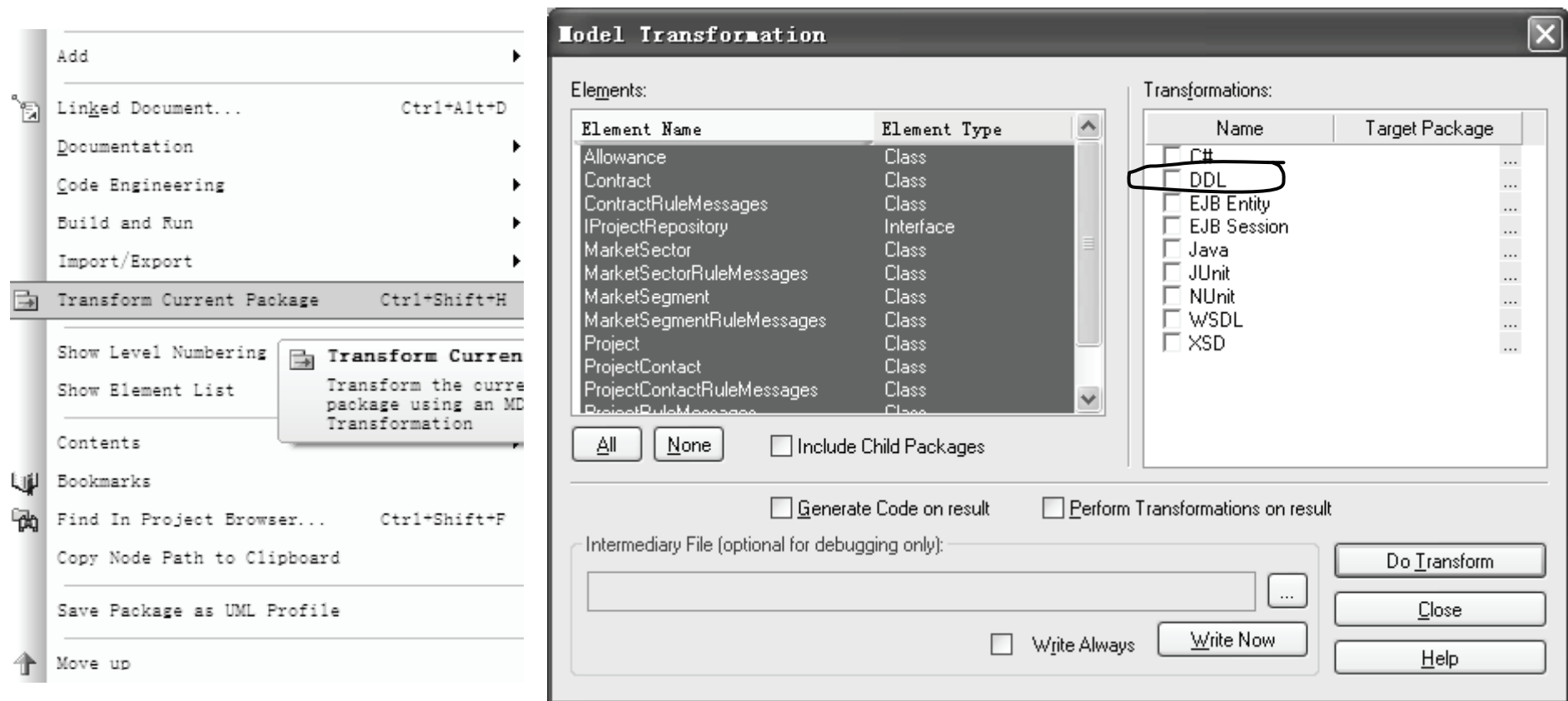


PowerDesigner

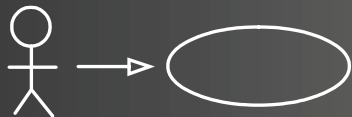


<http://www.umlchina.com>

建模工具自动映射



EA

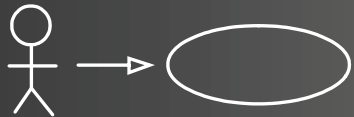


<http://www.umlchina.com>

数据层

 映射存储

 构造数据源层



数据源架构模式

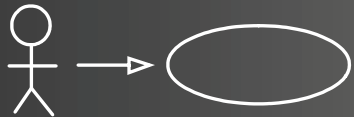
❖ 表数据入口

❖ 行数据入口

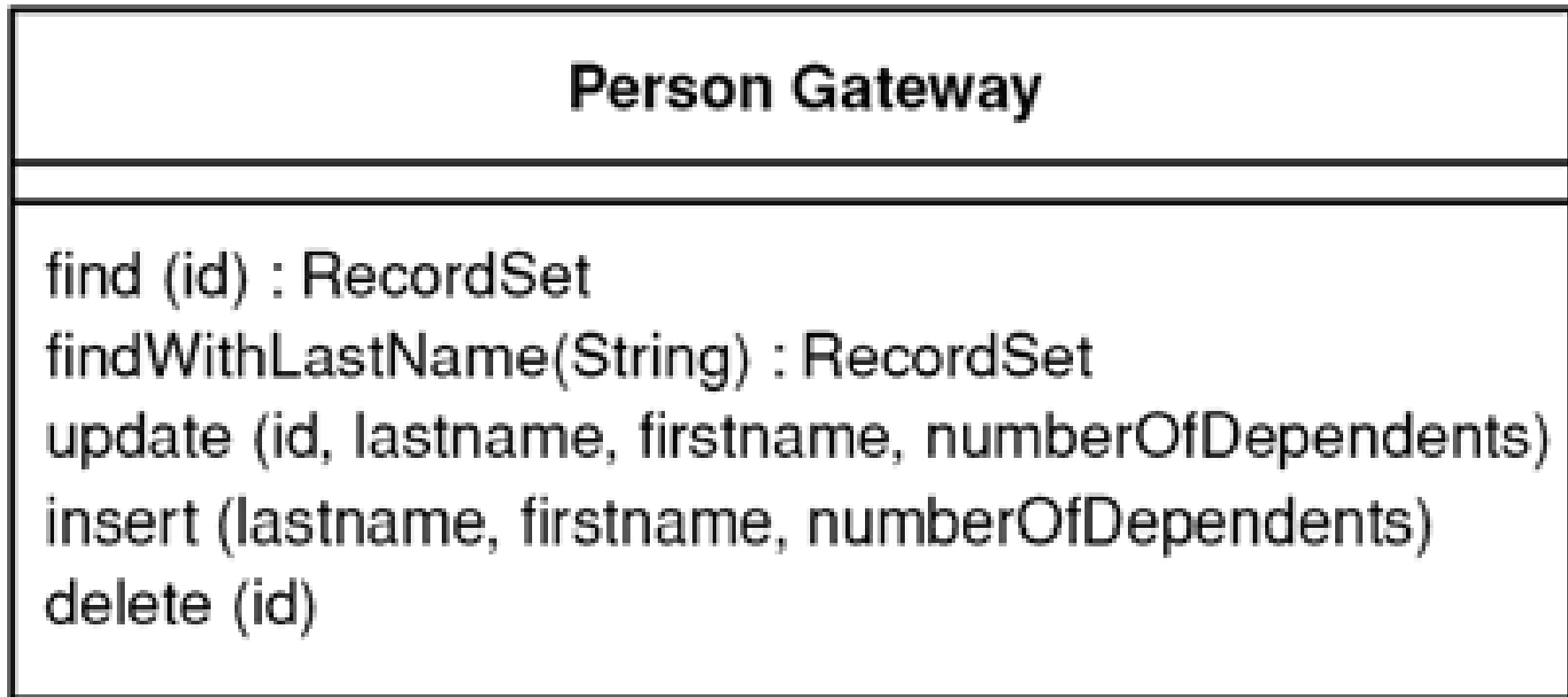
❖ 活动记录

❖ 数据映射器

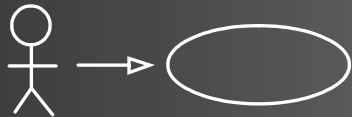
封装数据访问逻辑（SQL…）



表数据入口

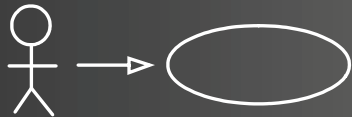


一个实例处理表中所有行



表数据入口

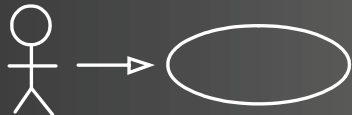
- ❖把所有和一张表有关的SQL放在同一个类
- ❖这个类只有一个对象
- ❖适用于表模块、事务脚本
- ❖操作
 - ❖findAll, findPerson, findWithAge, ...
 - ❖insert, delete, ...



表数据入口示例

```
class PersonGateway...

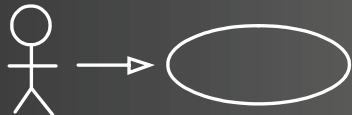
    public IDataReader FindAll() {
        String sql = "select * from person";
        return new OleDbCommand(sql, DB.Connection).ExecuteReader();
    }
    public IDataReader FindWithLastName(String lastName) {
        String sql = "SELECT * FROM person WHERE lastname = ?";
        IDbCommand comm = new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



表数据入口示例（续）

```
class PersonGateway...  
  
    public Object[] FindRow (long key) {  
        String sql = "SELECT * FROM person WHERE id = ?";  
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);  
        comm.Parameters.Add(new OleDbParameter("key",key));  
        IDataReader reader = comm.ExecuteReader();  
        reader.Read();  
        Object [] result = new Object[reader.FieldCount];  
        reader.GetValues(result);  
        reader.Close();  
        return result;  
    }
```

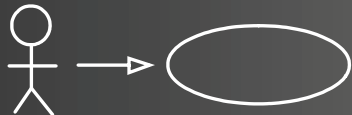
如果想一次读取多行



表数据入口示例（续）

```
class PersonGateway...  
  
    public void Update (long key, String lastname, String firstname, long  
➡ numberOfDependents){  
        String sql = @"  
            UPDATE person  
              SET lastname = ?, firstname = ?, numberOfDependents = ?  
             WHERE id = ?";  
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);  
        comm.Parameters.Add(new OleDbParameter ("last", lastname));  
        comm.Parameters.Add(new OleDbParameter ("first", firstname));  
        comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));  
        comm.Parameters.Add(new OleDbParameter ("key", key));  
        comm.ExecuteNonQuery();  
    }
```

更新

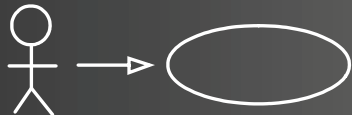


表数据入口示例（续）

```
class PersonGateway...

    public long Insert(String lastName, String firstName, long numberOfDependents) {
        String sql = "INSERT INTO person VALUES (?, ?, ?, ?)";
        long key = GetNextID();
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.Parameters.Add(new OleDbParameter ("last", lastName));
        comm.Parameters.Add(new OleDbParameter ("first", firstName));
        comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
        comm.ExecuteNonQuery();
        return key;
    }
```

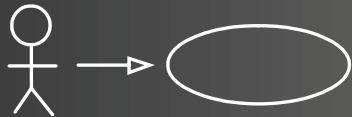
插入



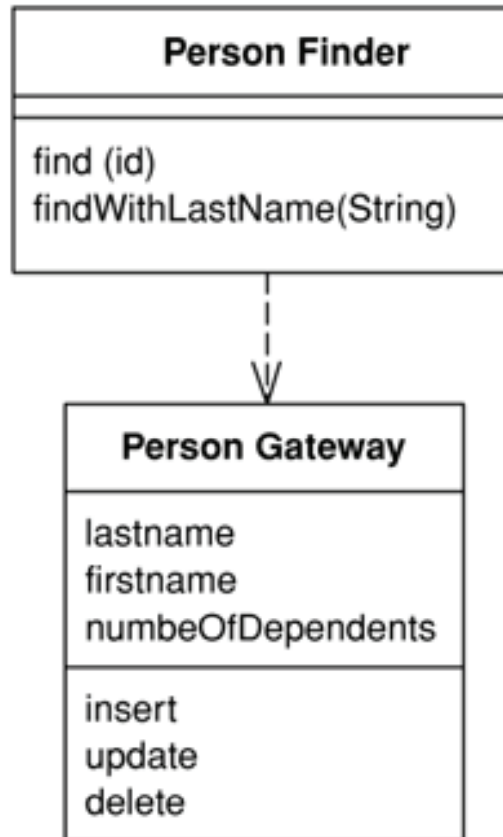
表数据入口示例（续）

```
class PersonGateway...  
  
    public void Delete (long key) {  
        String sql = "DELETE FROM person WHERE id = ?";  
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);  
        comm.Parameters.Add(new OleDbParameter ("key", key));  
        comm.ExecuteNonQuery();  
    }
```

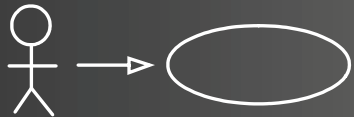
删除



行数据入口

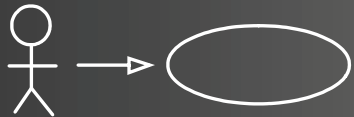


一行一个实例

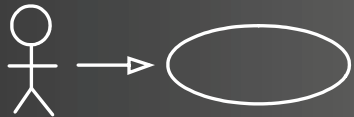
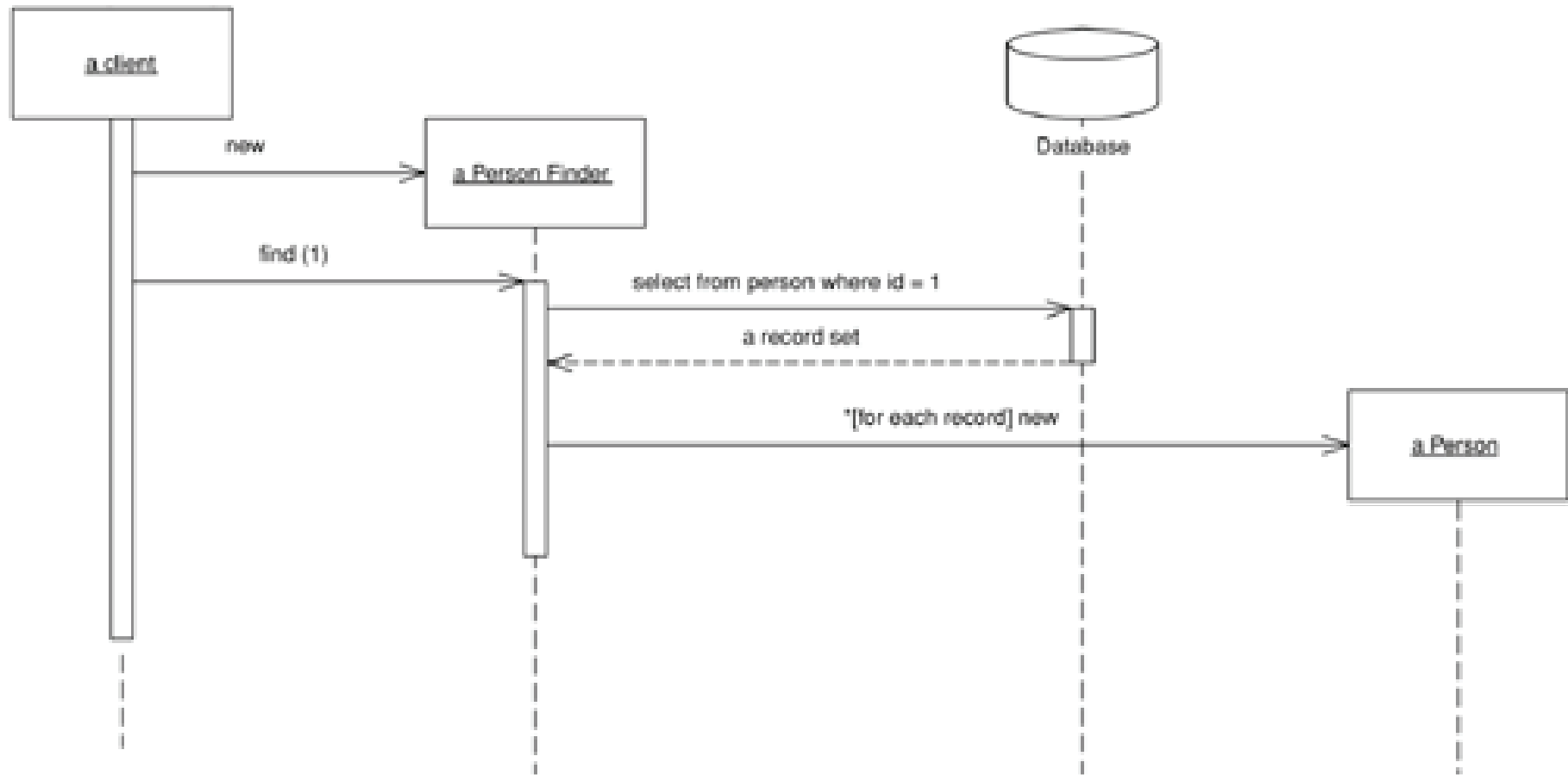


行数据入口

- ❖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



行数据入口交互

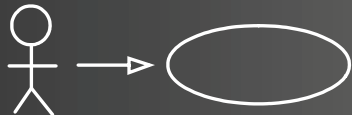


行数据入口示例

```
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

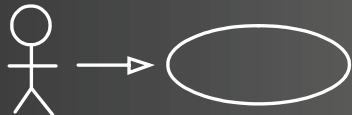


行数据入口示例（续）

```
class PersonGateway...

private static final String insertStatementString =
    "INSERT INTO people VALUES (?, ?, ?, ?)";
public Long insert() {
    PreparedStatement insertStatement = null;
    try {
        insertStatement = DB.prepare(insertStatementString);
        setID(findNextDatabaseId());
        insertStatement.setInt(1, getID().intValue());
        insertStatement.setString(2, lastName);
        insertStatement.setString(3, firstName);
        insertStatement.setInt(4, numberOfDependents);
        insertStatement.execute();
        Registry.addPerson(this);
        return getID();
    } catch (SQLException e) {
        throw new ApplicationException(e);
    } finally {DB.cleanUp(insertStatement);
    }
}
```

插入

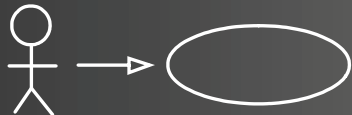


行数据入口示例（续）

```
class PersonGateway...

    public void Update (long key, String lastname, String firstname, long
➡ numberOfDependents){
        String sql = @"
            UPDATE person
              SET lastname = ?, firstname = ?, numberOfDependents = ?
              WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("last", lastname));
        comm.Parameters.Add(new OleDbParameter ("first", firstname));
        comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }
```

更新



行数据入口示例（续）

```
class PersonFinder...

    private final static String findStatementString =
        "SELECT id, lastname, firstname, number_of_dependents " +
        "  from people " +

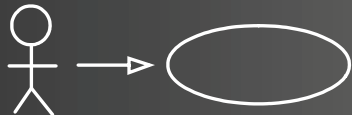
class PersonGateway...

    public static PersonGateway load(ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        PersonGateway result = (PersonGateway) Registry.getPerson(id);
        if (result != null) return result;
        String lastNameArg = rs.getString(2);
        String firstNameArg = rs.getString(3);
        int numDependentsArg = rs.getInt(4);
        result = new PersonGateway(id, lastNameArg, firstNameArg, numDependentsArg);
        Registry.addPerson(result);
        return result;
    }

    } finally {DB.cleanup(findStatement, rs);
    }
}

public PersonGateway find(long id) {
    return find(new Long(id));
}
```

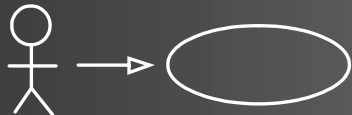
生成——通过Finder



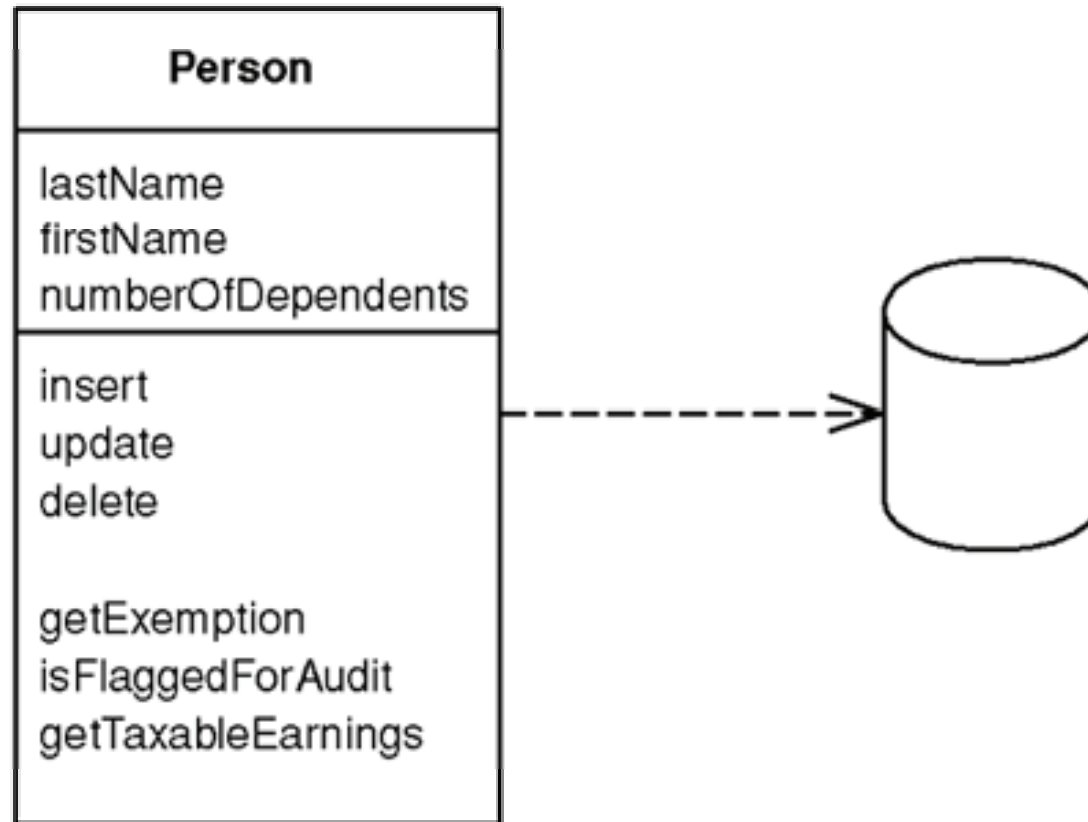
行数据入口示例（续）

```
PersonFinder finder = new PersonFinder();
Iterator people = finder.findResponsibles().iterator();
StringBuffer result = new StringBuffer();
while (people.hasNext()) {
    PersonGateway each = (PersonGateway) people.next();
    result.append(each.getLastName());
    result.append(" ");
    result.append(each.getFirstName());
    result.append(" ");
    result.append(String.valueOf(each.getNumberOfDependents()));
    result.append("
");
}
return result.toString();
```

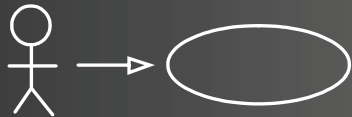
通过事务脚本使用行数据入口



活动记录

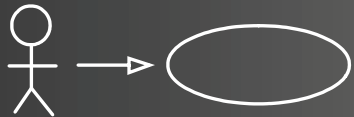


领域模型 + 数据逻辑



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



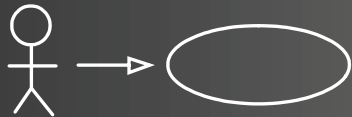
活动记录示例

```
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



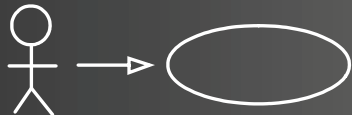
活动记录示例

```
class Person...

    private final static String findStatementString =
        "SELECT id, lastname, firstname, number_of_dependents" +
        " FROM people" +
        " WHERE id = ?";
public static Person load(ResultSet rs) throws SQLException {
    Long id = new Long(rs.getLong(1));
    Person result = (Person) Registry.getPerson(id);
    if (result != null) return result;
    String lastNameArg = rs.getString(2);
    String firstNameArg = rs.getString(3);
    int numDependentsArg = rs.getInt(4);
    result = new Person(id, lastNameArg, firstNameArg, numDependentsArg);
    Registry.addPerson(result);
    return result;
}

    } catch (SQLException e) {
        throw new ApplicationException(e);
    } finally {
        DB.cleanUp(findStatement, rs);
    }
}
```

查找和加载

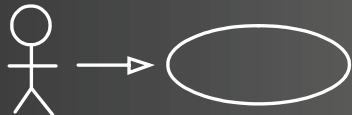


活动记录示例（续）

```
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

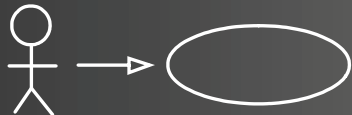


活动记录示例（续）

```
class Person...

private final static String insertStatementString =
    "INSERT INTO people VALUES (?, ?, ?, ?)";
public Long insert() {
    PreparedStatement insertStatement = null;
    try {
        insertStatement = DB.prepare(insertStatementString);
        setID(findNextDatabaseId());
        insertStatement.setInt(1, getID().intValue());
        insertStatement.setString(2, lastName);
        insertStatement.setString(3, firstName);
        insertStatement.setInt(4, numberOfDependents);
        insertStatement.execute();
        Registry.addPerson(this);
        return getID();
    } catch (Exception e) {
        throw new ApplicationException(e);
    } finally {
        DB.cleanUp(insertStatement);
    }
}
```

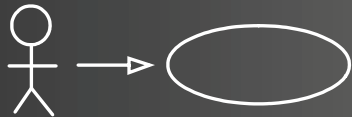
插入



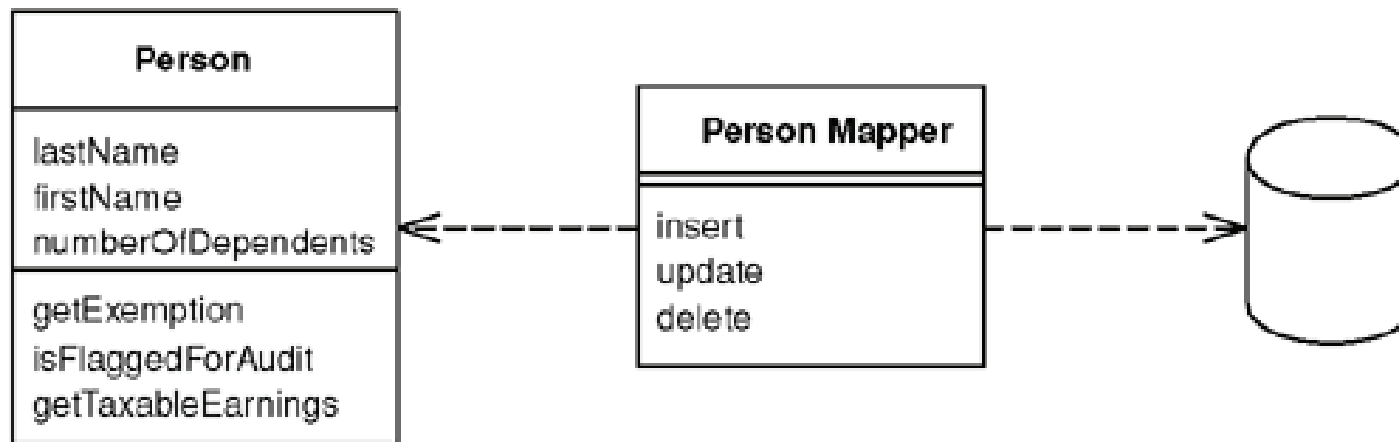
活动记录示例（续）

```
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



数据映射器

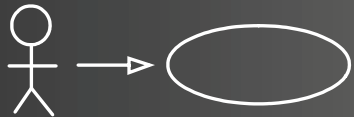


保持对象和数据库独立

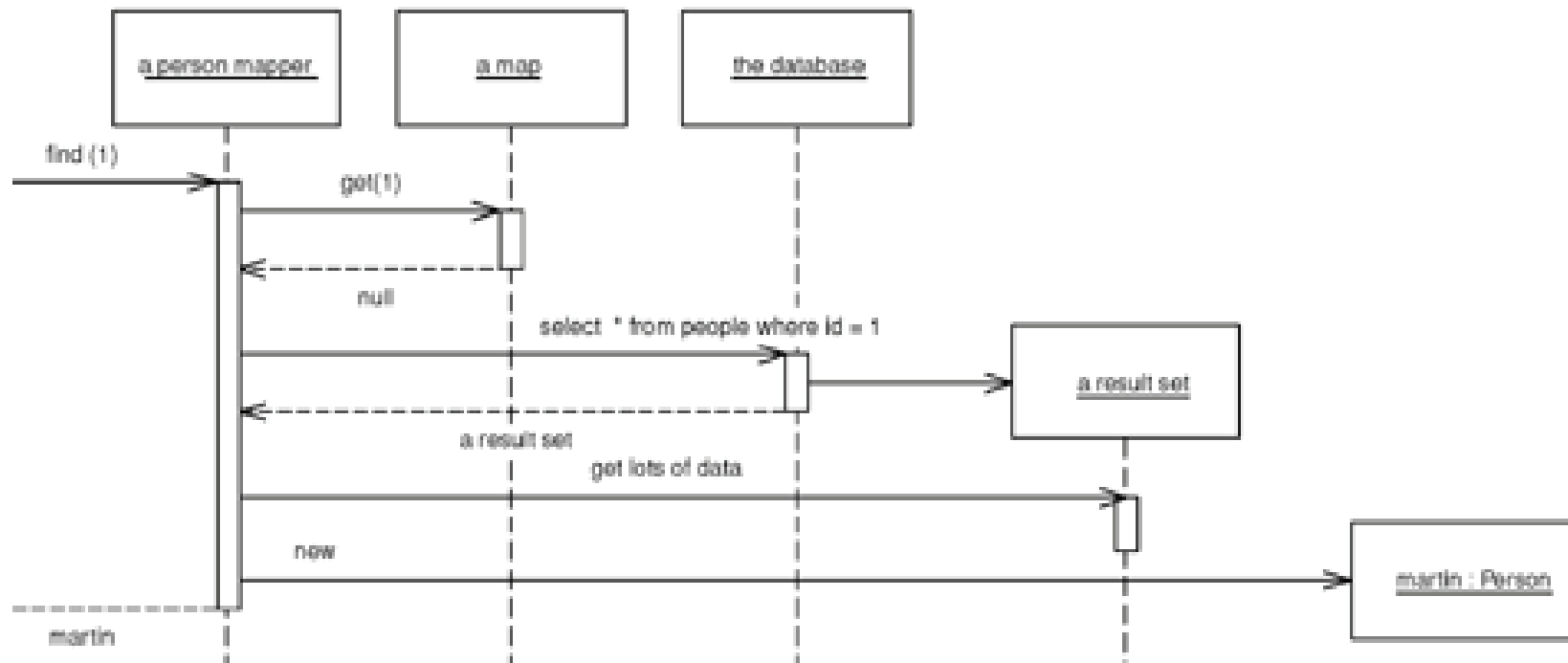


数据映射器

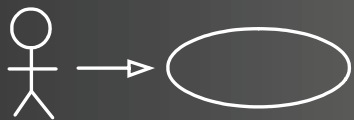
- ❖ 在领域模型和数据库之间移动数据
- ❖ 领域模型基本不知道数据映射器
- ❖ 数据库设计独立于领域模型
- ❖ 和领域模型一起使用
- ❖ 复杂，手工创建不如购买



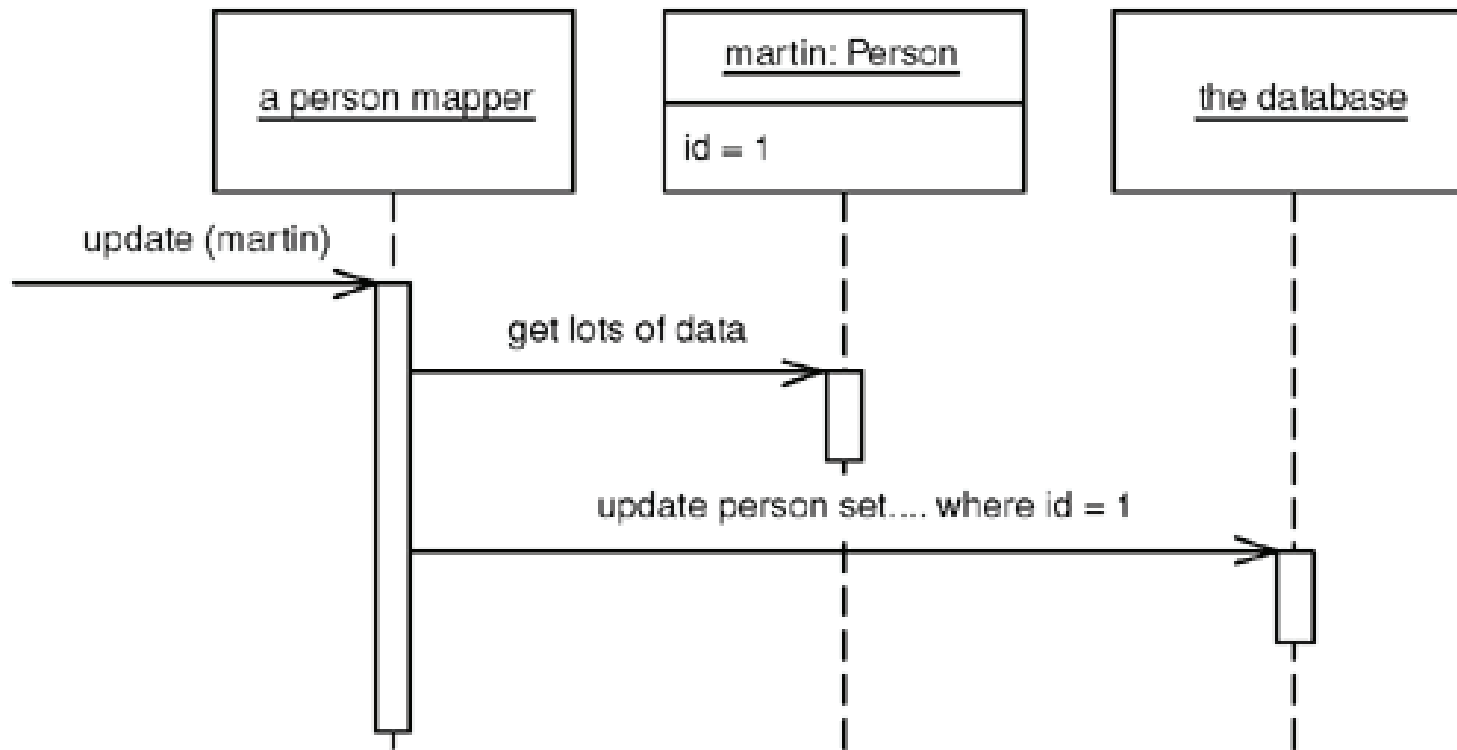
数据映射器交互



出来——从数据库读取数据



数据映射器交互



进去——更新数据



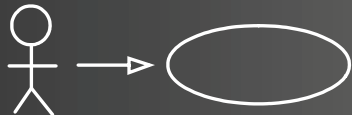
数据映射器示例

```
class AbstractMapper...

protected Map loadedMap = new HashMap();
abstract protected String findStatement();
protected DomainObject abstractFind(Long id) {
    DomainObject result = (DomainObject) loadedMap.get(id);
    if (result != null) return result;
    PreparedStatement findStatement = null;
    try {
        findStatement = DB.prepare(findStatement());
        findStatement.setLong(1, id.longValue());
        ResultSet rs = findStatement.executeQuery();           _dependents ";
        rs.next();
        result = load(rs);
        return result;
    } catch (SQLException e) {
        throw new ApplicationException(e);
    } finally {
        DB.cleanUp(findStatement);
    }
}

class PersonMe
protected S
    return "
        " FF
        " WE
}
public stat
public Pers
    return (
}
public Pers
    return f
}
```

查找和加载



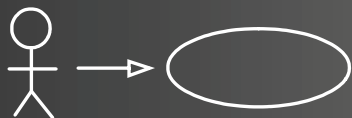
数据映射器示例（续）

```
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



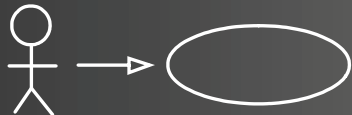
数据映射器示例（续）

```
class AbstractMapper...

public Long insert(DomainObject subject) {
    PreparedStatement insertStatement = null;
    try {
        insertStatement = DB.prepare(insertStatement());
        subject.setID(findNextDatabaseId());
        insertStatement.setInt(1, subject.getID().intValue());
        doInsert(subject, insertStatement);
        insertStatement.execute();
        loadedMap.put(subject.getID(), subject);
        return subject.getID();
    } catch (SQLException e) {
        throw new ApplicationException(e);
    } finally {
        DB.cleanUp(insertStatement);
    }
}

abstract protected String insertStatement();
abstract protected void doInsert(DomainObject subject, PreparedStatement
insertStatement)
    throws SQLException;
```

插入



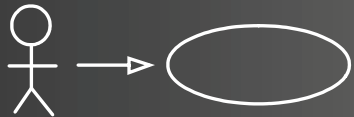
业务层模式

❖ 领域模型

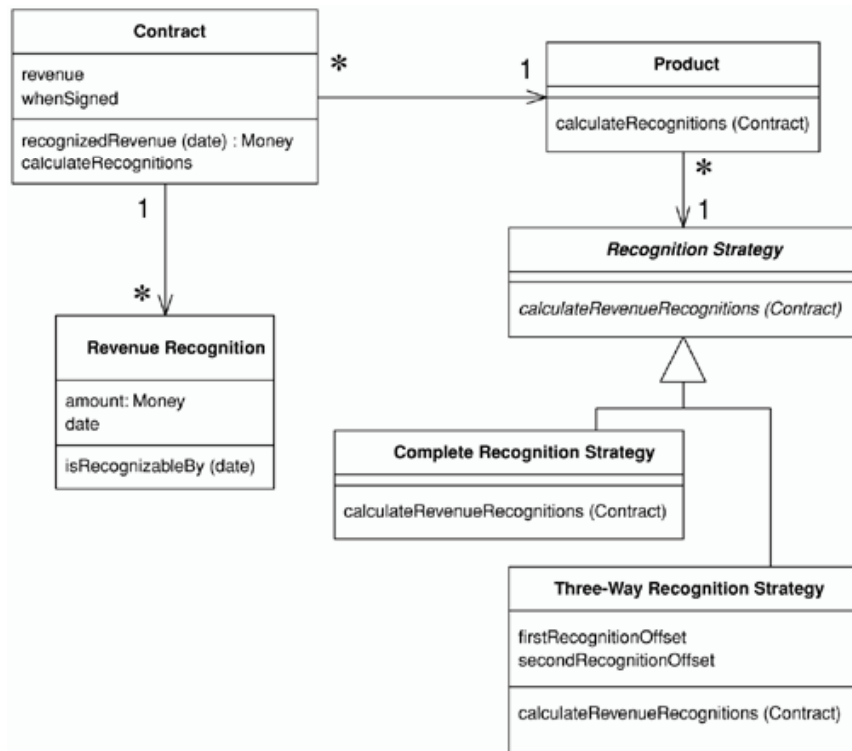
❖ 表模块

❖ 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

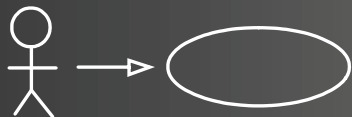
❖ 每个领域对象对自己的状态和函数负责

❖ 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）

❖ 例子：实体Bean，对象映射工具例如JDO、Torque或甚至POJO实现



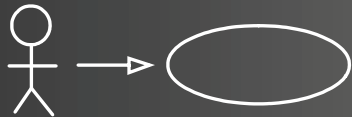
领域模型

❖ 优点

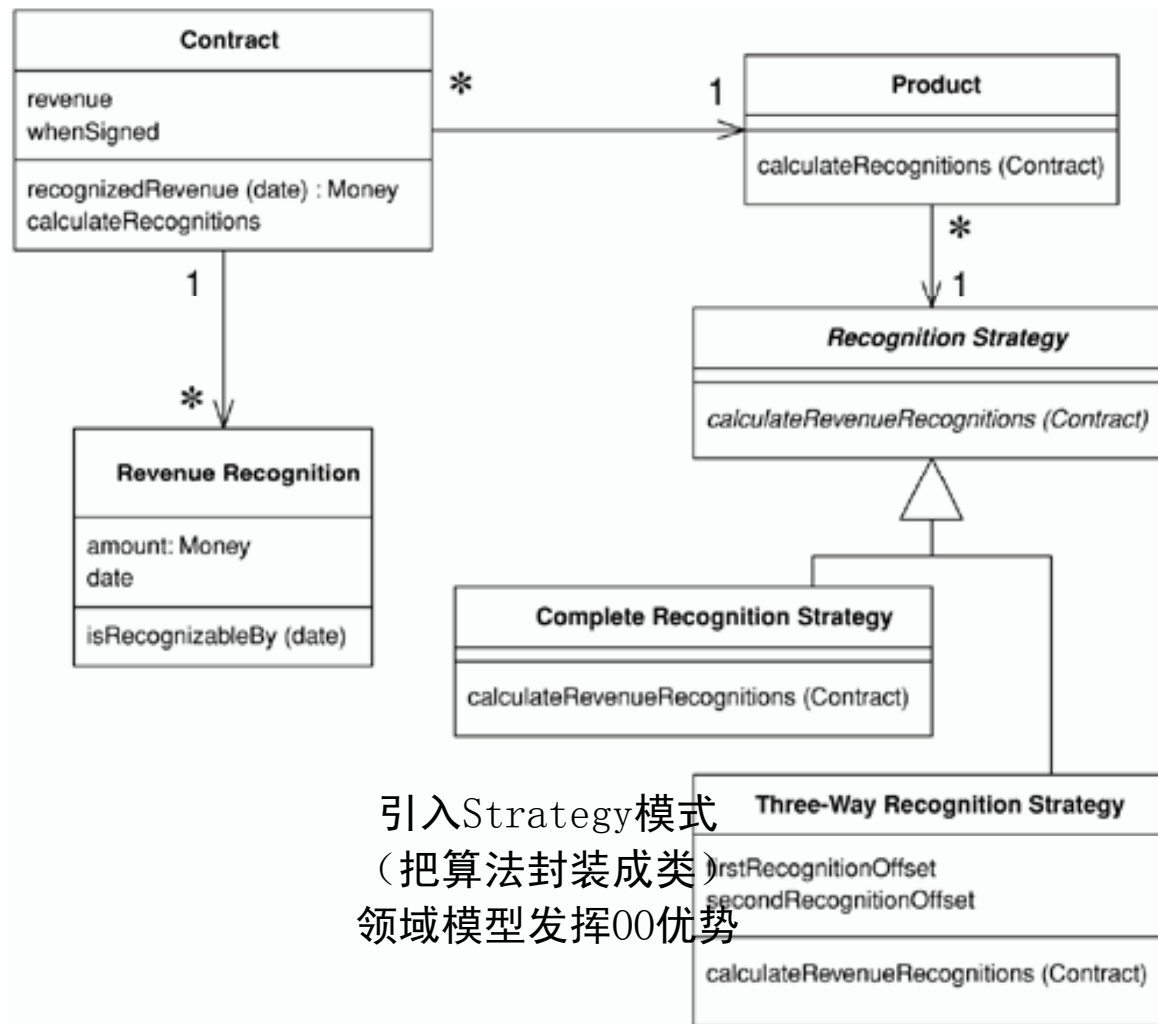
- ❖ 可以以OO的方式为复杂业务逻辑建模
- ❖ 使用关系对象映射工具支持自动化生成
- ❖ 能够维护与其他领域对象的关系

❖ 缺点

- ❖ 大量对象
- ❖ 和另一个领域对象的关系会太复杂
- ❖ 在web情况下，处理事务变难

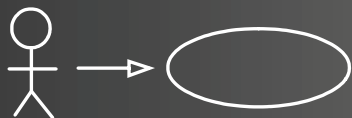


领域模型示例



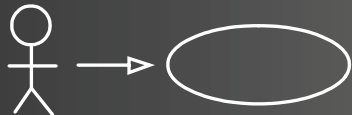
类图

引入Strategy模式
(把算法封装成类)
领域模型发挥OO优势



领域模型示例（续）

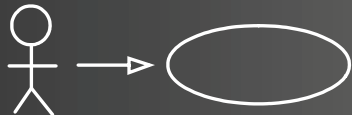
```
class RevenueRecognition...  
  
    private Money amount;  
    private MfDate date;  
    public RevenueRecognition(Money amount, MfDate date) {  
        this.amount = amount;  
        this.date = date;  
    }  
    public Money getAmount() {  
        return amount;  
    }  
    boolean isRecognizableBy(MfDate asOf) {  
        return asOf.after(date) || asOf.equals(date);  
    }  
}
```



领域模型示例（续）

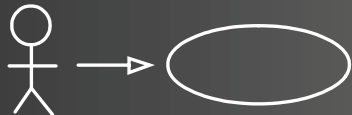
```
class Contract...
```

```
private List revenueRecognitions = new ArrayList();  
public Money recognizedRevenue(MfDate asOf) {  
    Money result = Money.dollars(0);  
    Iterator it = revenueRecognitions.iterator();  
    while (it.hasNext()) {  
        RevenueRecognition r = (RevenueRecognition) it.next();  
        if (r.isRecognizableBy(asOf))  
            result = result.add(r.getAmount());  
    }  
    return result;  
}
```



领域模型示例（续）

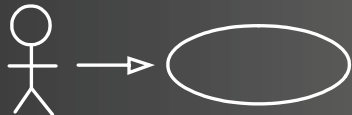
```
class Contract...  
  
    private Product product;  
    private Money revenue;  
    private MfDate whenSigned;  
    private Long id;  
    public Contract(Product product, Money revenue, MfDate whenSigned) {  
        this.product = product;  
        this.revenue = revenue;  
        this.whenSigned = whenSigned;  
    }
```



领域模型示例（续）

```
class Product...

    private String name;
    private RecognitionStrategy recognitionStrategy;
    public Product(String name, RecognitionStrategy recognitionStrategy) {
        this.name = name;
        this.recognitionStrategy = recognitionStrategy;
    }
    public static Product newWordProcessor(String name) {
        return new Product(name, new CompleteRecognitionStrategy());
    }
    public static Product newSpreadsheet(String name) {
        return new Product(name, new ThreeWayRecognitionStrategy(60, 90));
    }
    public static Product newDatabase(String name) {
        return new Product(name, new ThreeWayRecognitionStrategy(30, 60));
    }
}
```



领域模型示例（续）

```
class RecognitionStrategy...
```

```
    abstract void calculateRevenueRecognitions(Contract contract);
```

```
class CompleteRecognitionStrategy...
```

```
    void calculateRevenueRecognitions(Contract contract) {
```

```
        contract.addRevenueRecognition(new RevenueRecognition(contract.getRevenue(),
```

```
        ➡ contract.getWhenSigned()));
```

```
    }
```

```
class ThreeWayRecognitionStrategy...
```

```
    private int firstRecognitionOffset;
```

```
    private int secondRecognitionOffset;
```

```
    public ThreeWayRecognitionStrategy(int firstRecognitionOffset,  
                                       int secondRecognitionOffset)
```

```
    {
```

```
        this.firstRecognitionOffset = firstRecognitionOffset;
```

```
        this.secondRecognitionOffset = secondRecognitionOffset;
```

```
    }
```

```
    void calculateRevenueRecognitions(Contract contract) {
```

```
        Money[] allocation = contract.getRevenue().allocate(3);
```

```
        contract.addRevenueRecognition(new RevenueRecognition  
            (allocation[0], contract.getWhenSigned()));
```

```
        contract.addRevenueRecognition(new RevenueRecognition
```

```
            (allocation[1], contract.getWhenSigned().addDays(firstRecognitionOffset)));
```

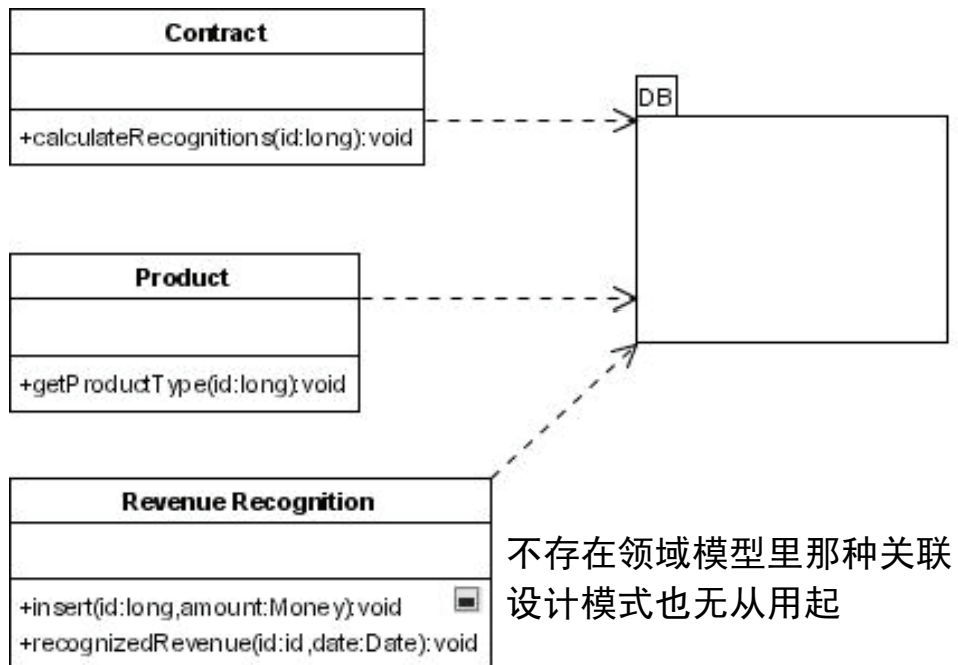
```
        contract.addRevenueRecognition(new RevenueRecognition
```

```
            (allocation[2], contract.getWhenSigned().addDays(secondRecognitionOffset)));
```

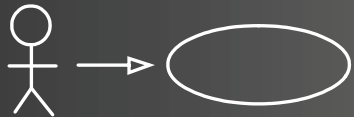
```
    }
```



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
- ❖ 可以
- ❖ 例子: .NET、POJO、Java Resultset



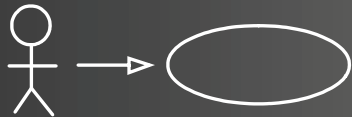
表模块

❖ 优点

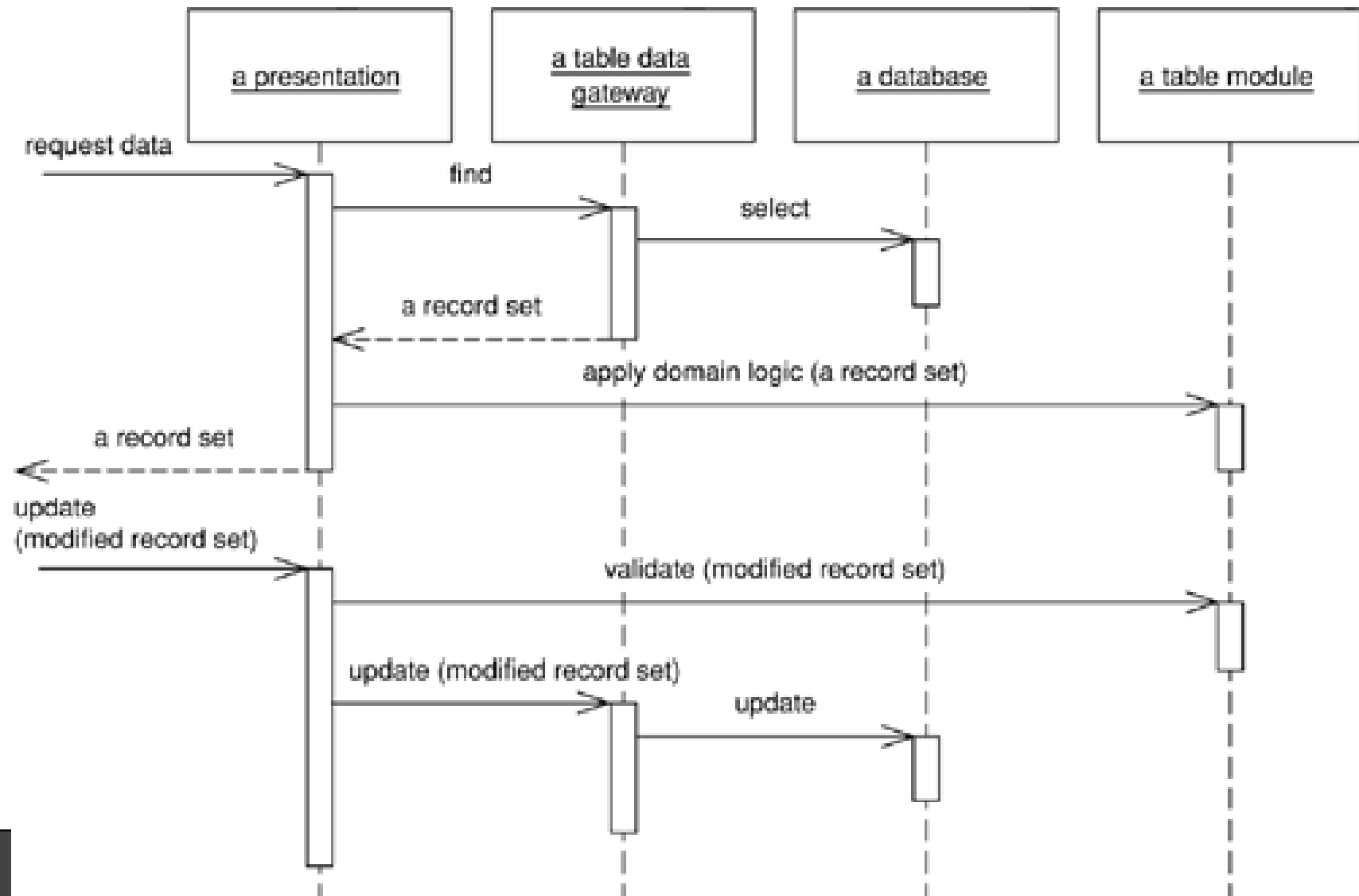
- ❖ 和使用记录集的框架配合良好
- ❖ 和数据库表更接近
- ❖ 通过在表模块中书写方法，可以检索复杂记录集
- ❖ 可以返回能被一致调用的记录集

❖ 缺点

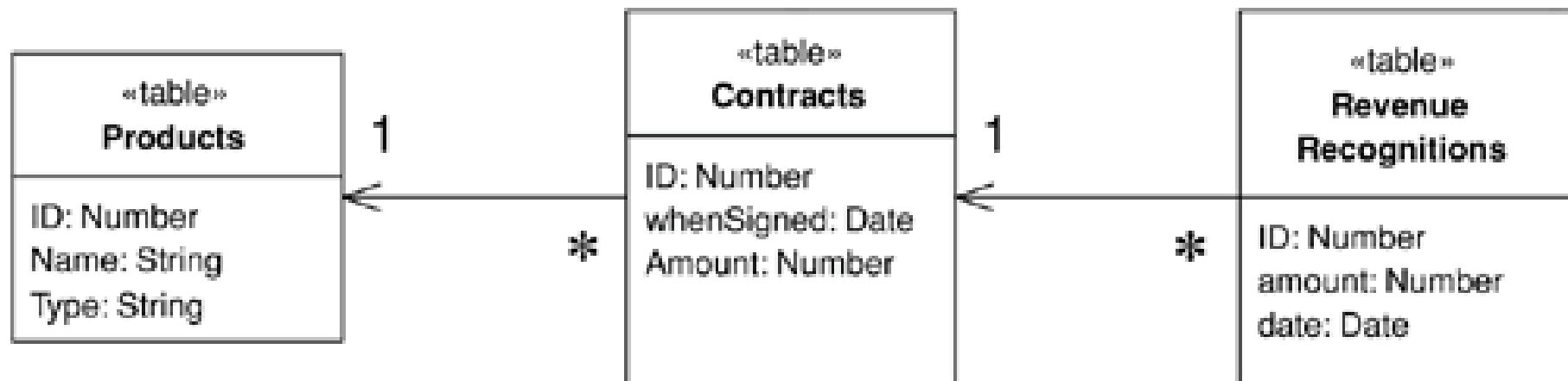
- ❖ 依赖于潜在的实现
- ❖ 类型只能细到记录集，不支持更高级别（对象类型检查）
- ❖ 没有发挥OO设计的全部能量



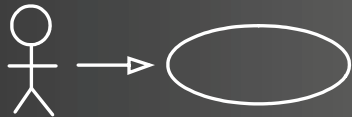
表模块典型交互



表模块示例



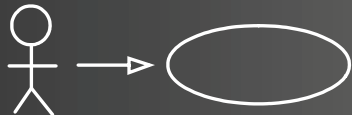
数据库结构



表模块示例（续）

```
class TableModule...  
  
    protected DataTable table;  
    protected TableModule(DataSet ds, String tableName) {  
        table = ds.Tables[tableName];  
    }
```

表模块超类



表模块示例（续）

```
class Contract...
```

```
public Contract (DataSet ds) : base (ds, "Contracts") {}
```

以表名为参数调用超类构造函数

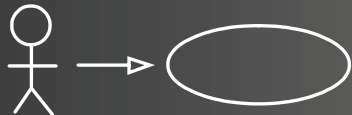
```
contract = new Contract (dataset);
```

通过传入数据集创建表模块对象

```
class Contract...
```

```
public DataRow this [long key] {  
    get {  
        String filter = String.Format("ID = {0}", key);  
        return table.Select(filter)[0];  
    }  
}
```

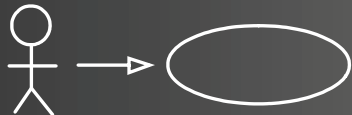
索引器寻找特定行



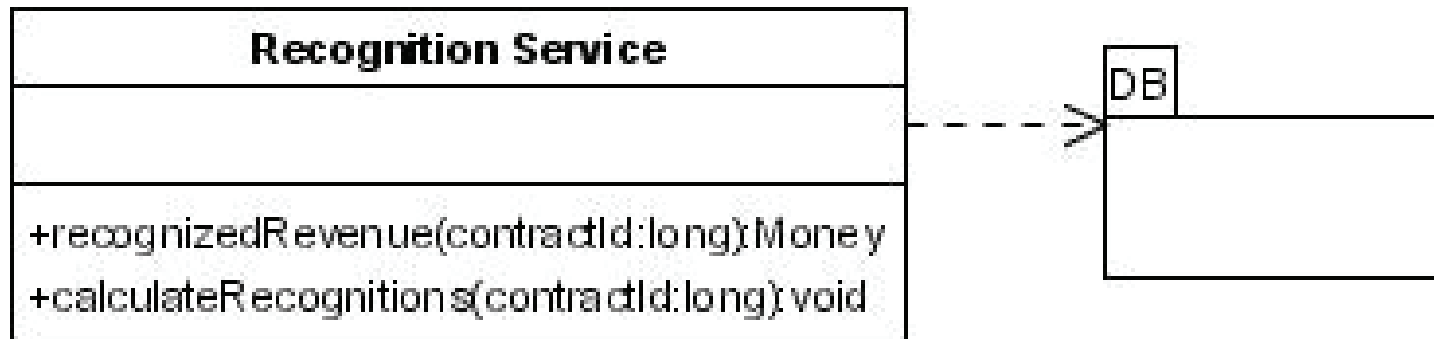
表模块示例（续）

```
class Contract...
```

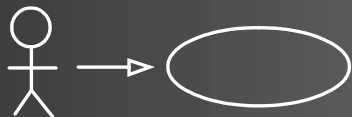
```
    public void CalculateRecognitions (long contractID) {
        DataRow contractRow = this[contractID];
        Decimal amount = (Decimal)contractRow["amount"];
        RevenueRecognition rr = new RevenueRecognition (table.DataSet);
        Product prod = new Product(table.DataSet);
        long 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");
    }
```



事务脚本



- ❖ 操作按过程调用组织
- ❖ 类似于函数编程，但可以应用OO技术来避免重复代码
- ❖ 可使用Command和/或Strategy模式实现
- ❖ 操作有明显的边界
- ❖ 例子：cgi、Java Servlet



事务脚本

❖ 优点

- ❖ KISS (Keep It Simple Stupid)

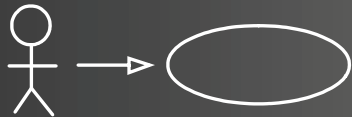
- ❖ 可以把事务放在功能边界中

❖ 缺点

- ❖ 难以处理复杂逻辑

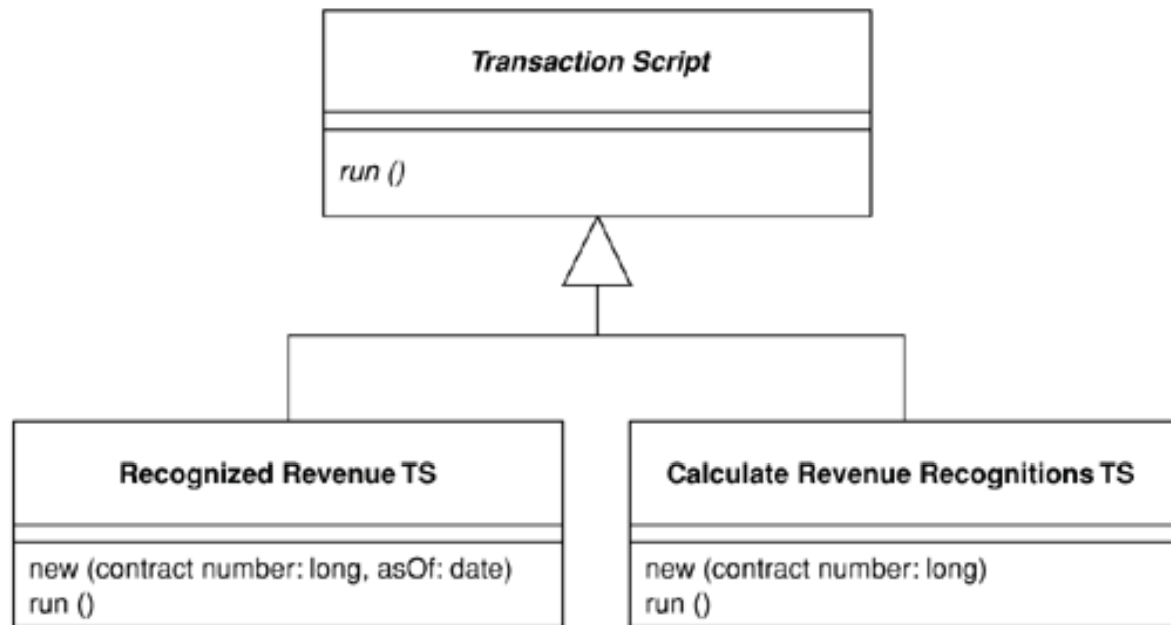
- ❖ 过程会很快增加

- ❖ 代码重复（通过Command模式可以消除部分）

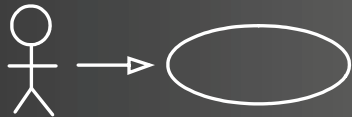


事务脚本

- ❖ 每个类（按主题）组织一些脚本
- ❖ 每个事务脚本一个类
 - ❖ 使用Command模式



可以组织到类中

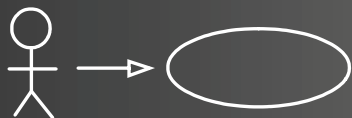


事务脚本示例



```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn date,
                                   PRIMARY KEY (contract, recognizedOn))
```

收入确认案例的类图和表



<http://www.umlchina.com>

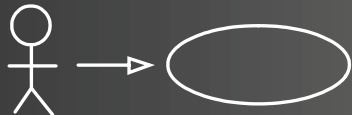
事务脚本示例（续）

```
class Gateway...

public ResultSet findRecognitionsFor(long contractID, MfDate asof) throws SQLException{
    PreparedStatement stmt = db.prepareStatement(findRecognitionsStatement);
    stmt = db.prepareStatement(findRecognitionsStatement);
    stmt.setLong(1, contractID);
    stmt.setDate(2, asof.toSqlDate());
    ResultSet result = stmt.executeQuery();
    return result;
}

private static final String findRecognitionsStatement =
    "SELECT amount " +
    "FROM revenueRecognitions " +
    "WHERE contract = ? AND recognizedOn <= ?";
private Connection db;
```

数据入口



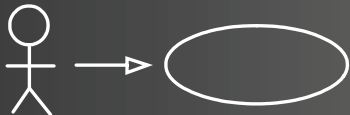
<http://www.umlchina.com>

事务脚本示例（续）

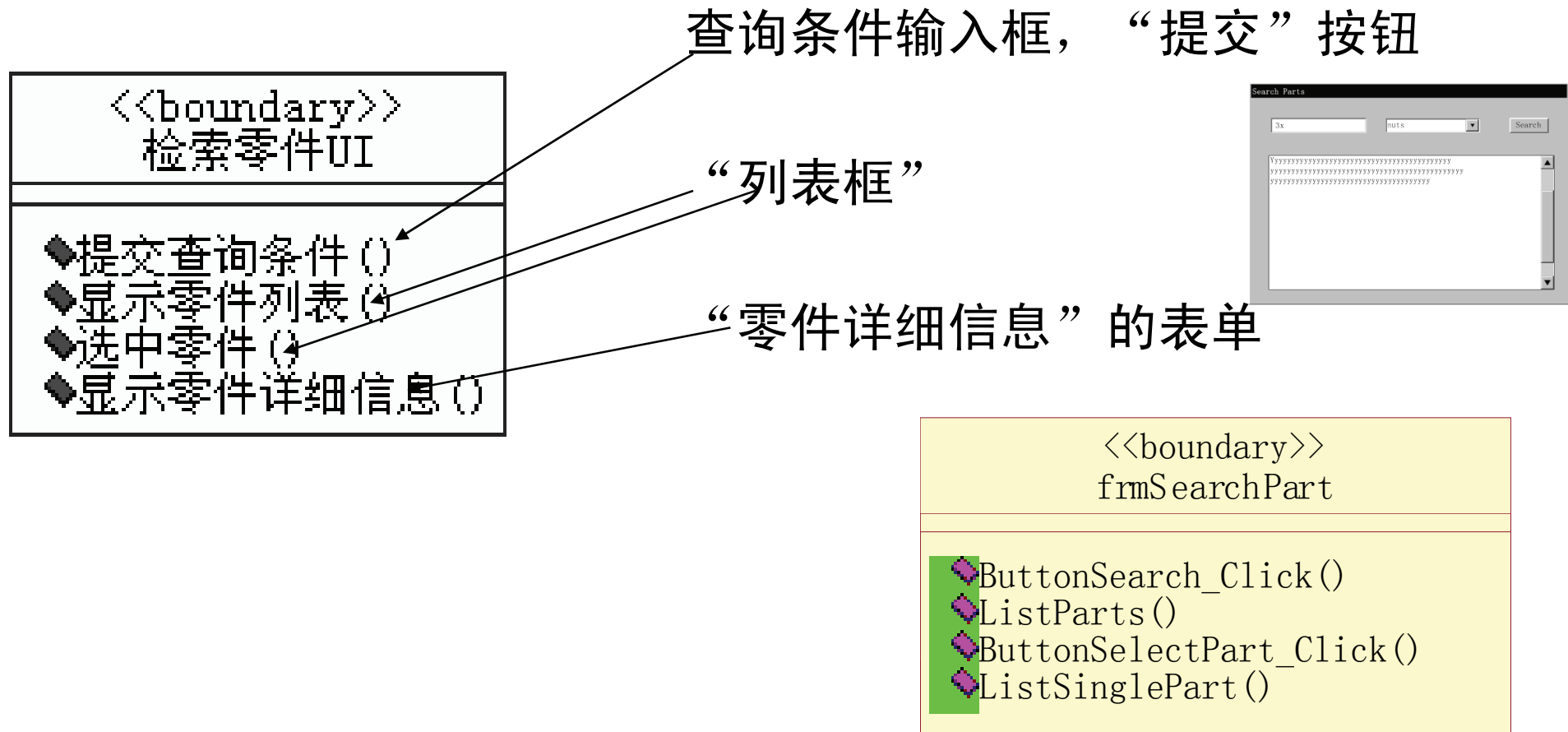
```
class RecognitionService...

    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);
        }
    }
}
```

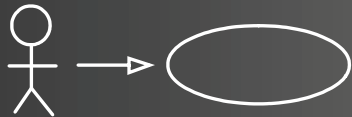
事务脚本



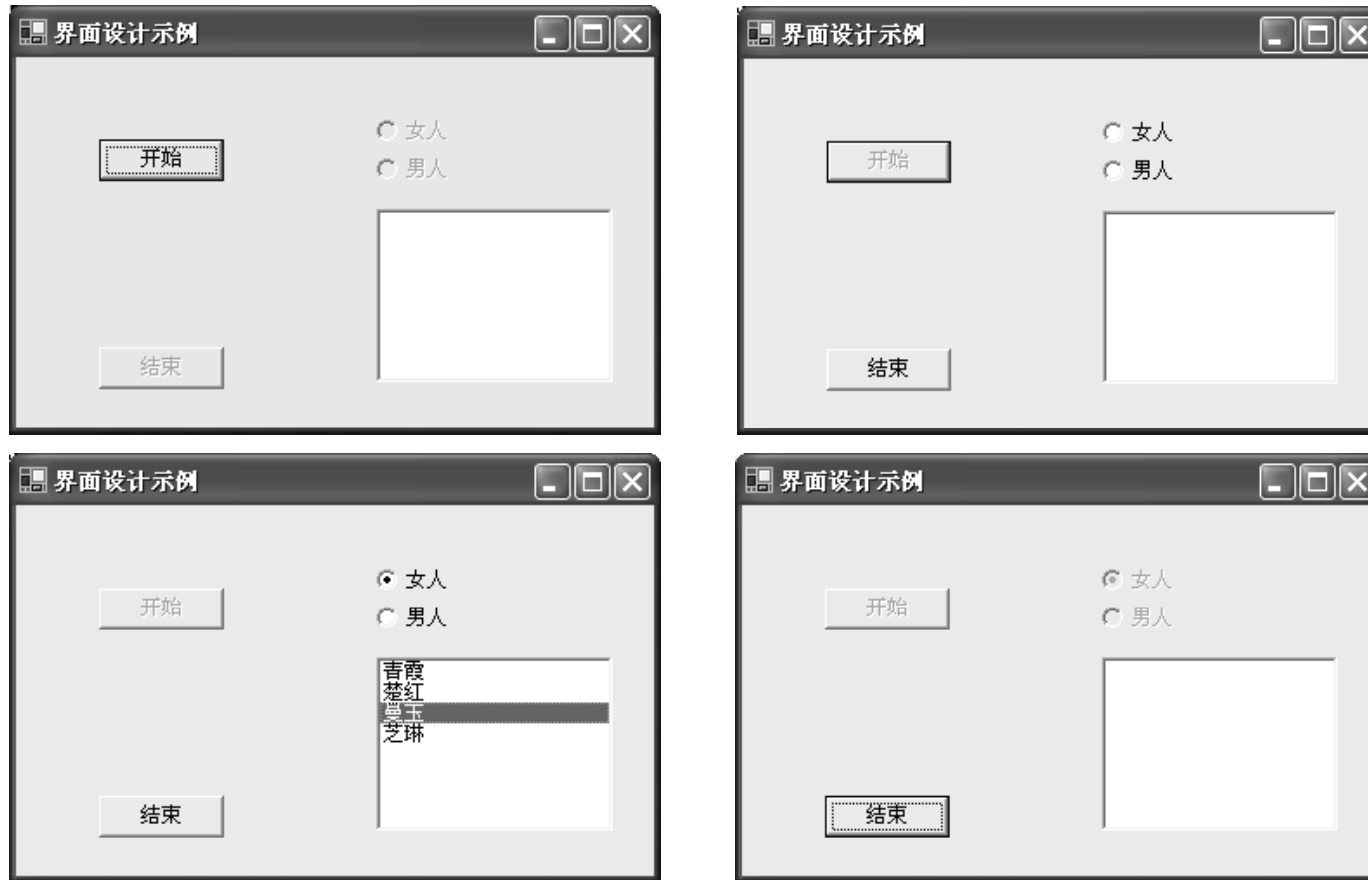
界面层



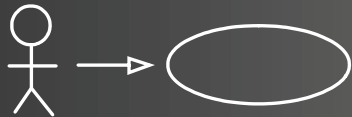
最简洁的界面：刚好能履行分析所赋予的责任



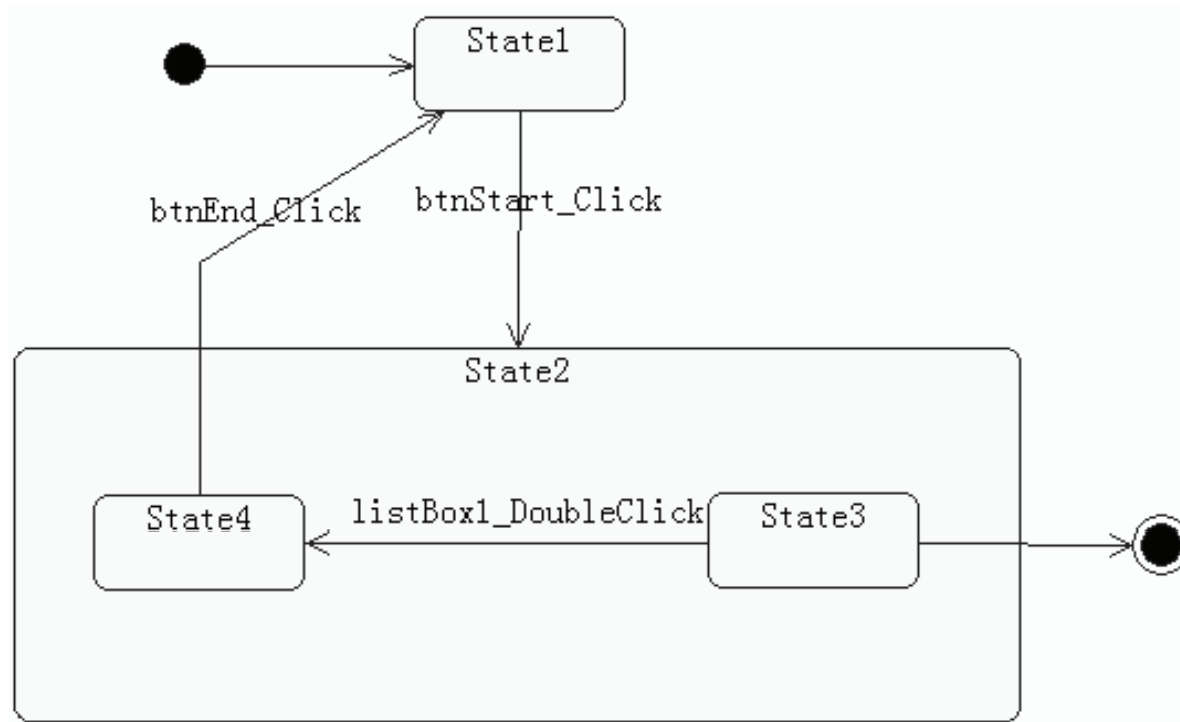
界面类状态图辅助设计



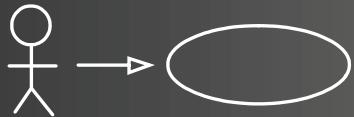
界面类接收大量消息，不断改变状态



界面类状态图辅助设计



以状态图主导界面设计



界面

功能需求—>可靠性需求—>可用性需求



界面正确

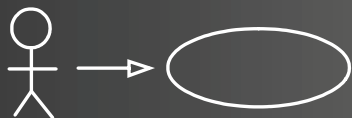


界面稳定



界面高效

交互设计越来越重要



Web表现模式

❖模型—视图—控制器（基本思想）

❖控制器

❖页面控制器

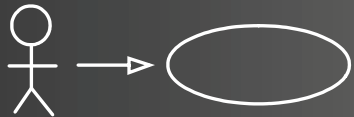
❖前端控制器

❖视图

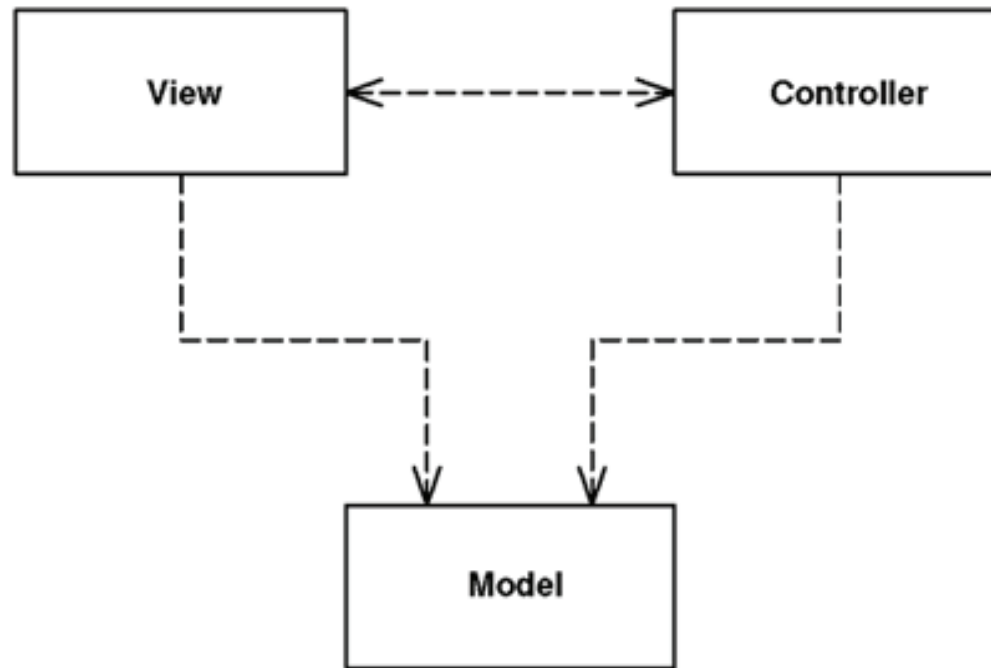
❖模板视图

❖转换视图

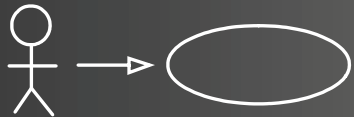
❖两步视图



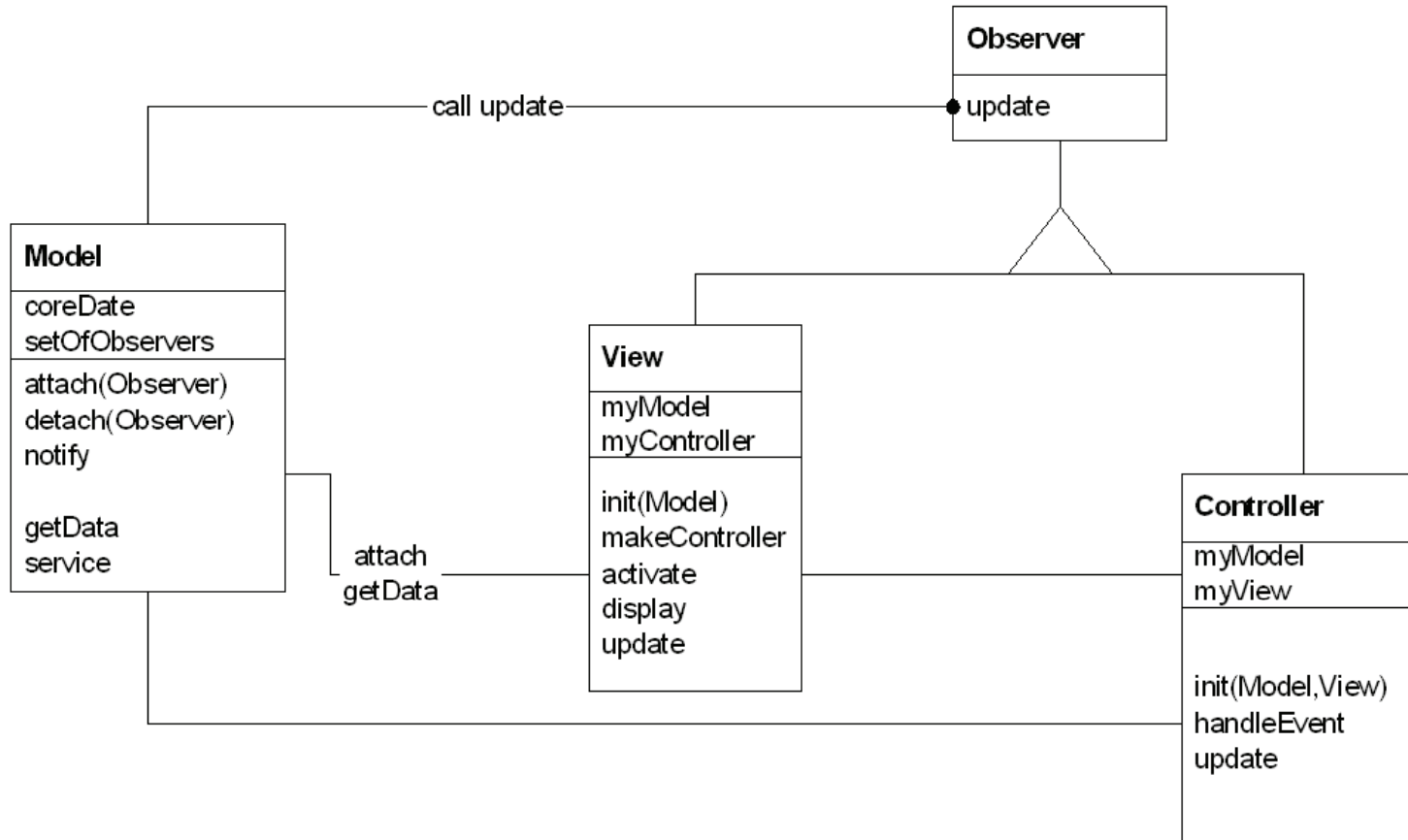
模型—视图—控制器



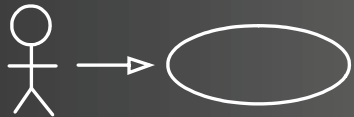
把用户界面交互分拆到三种不同角色



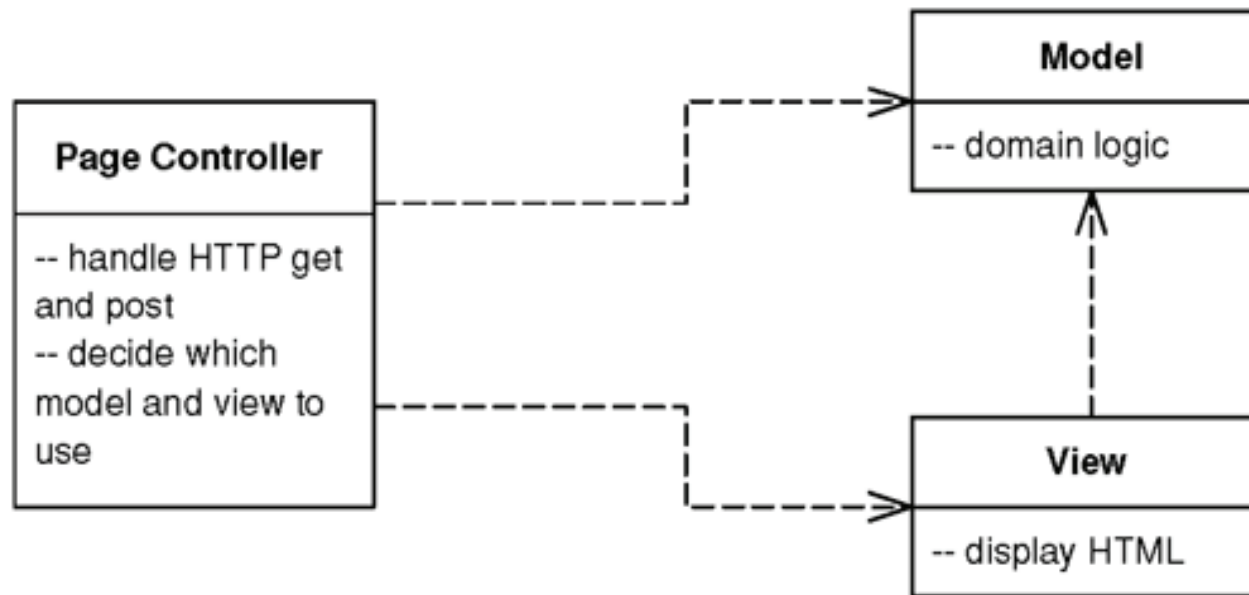
模型—视图—控制器



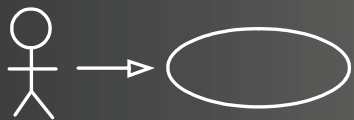
引入观察者



页面控制器

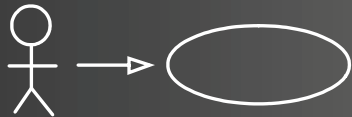


为每一个逻辑页面准备一个控制器

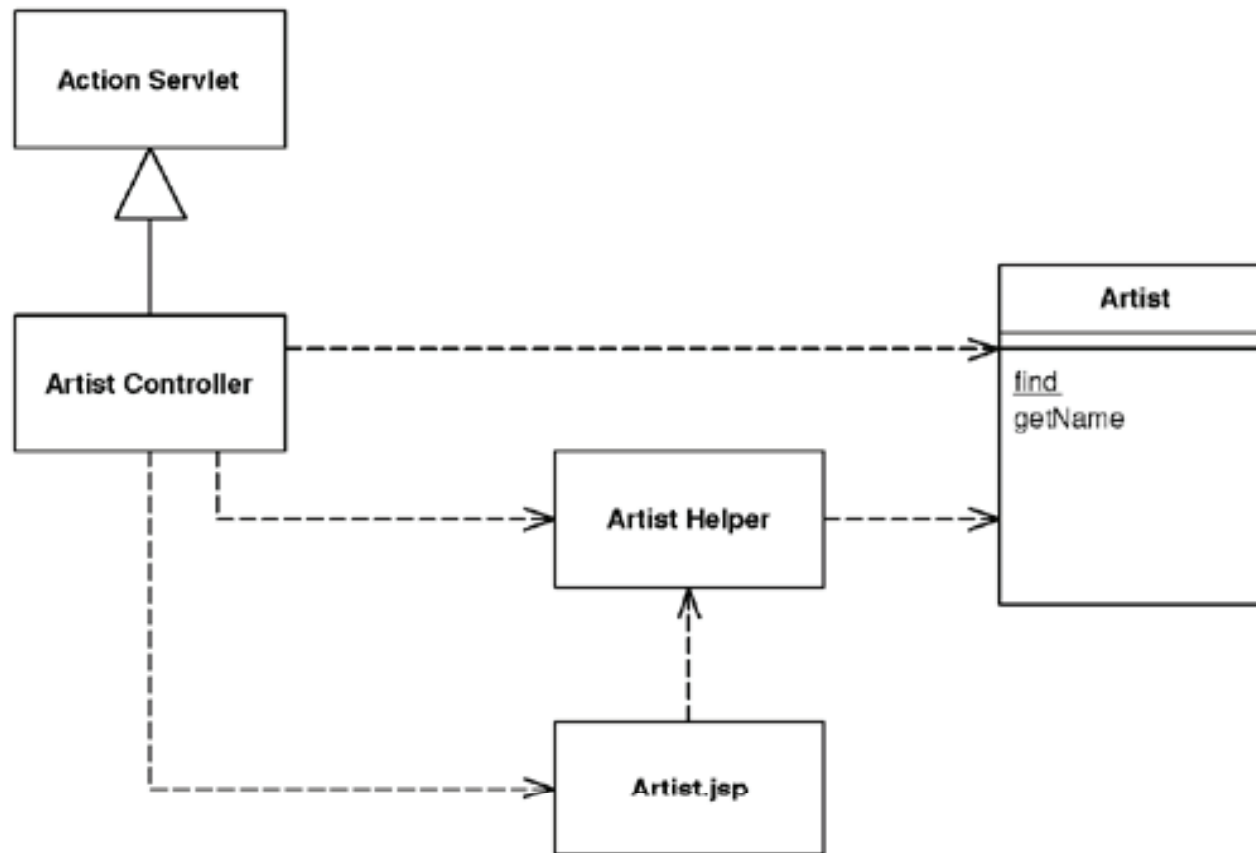


页面控制器

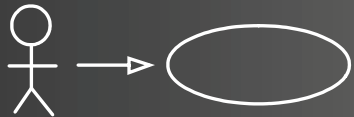
- ❖ 解码URL
- ❖ 创建并调用模型对象
- ❖ 决定哪个视图来显示
- ❖ 优点
 - ❖ 实现简单
 - ❖ 简单控制逻辑适用
 - ❖ 简单导航适用
- ❖ 缺点
 - ❖ 重复代码



页面控制器示例



类图

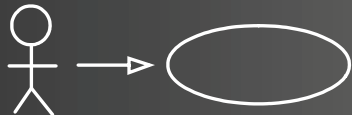


页面控制器示例（续）

```
class ArtistController...

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        if (artist == null)
            forward("/MissingArtistError.jsp", request, response);
        else {
            request.setAttribute("helper", new ArtistHelper(artist));
            forward("/artist.jsp", request, response);
        }
    }
}
```

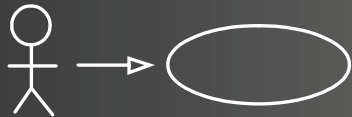
处理请求



页面控制器示例（续）

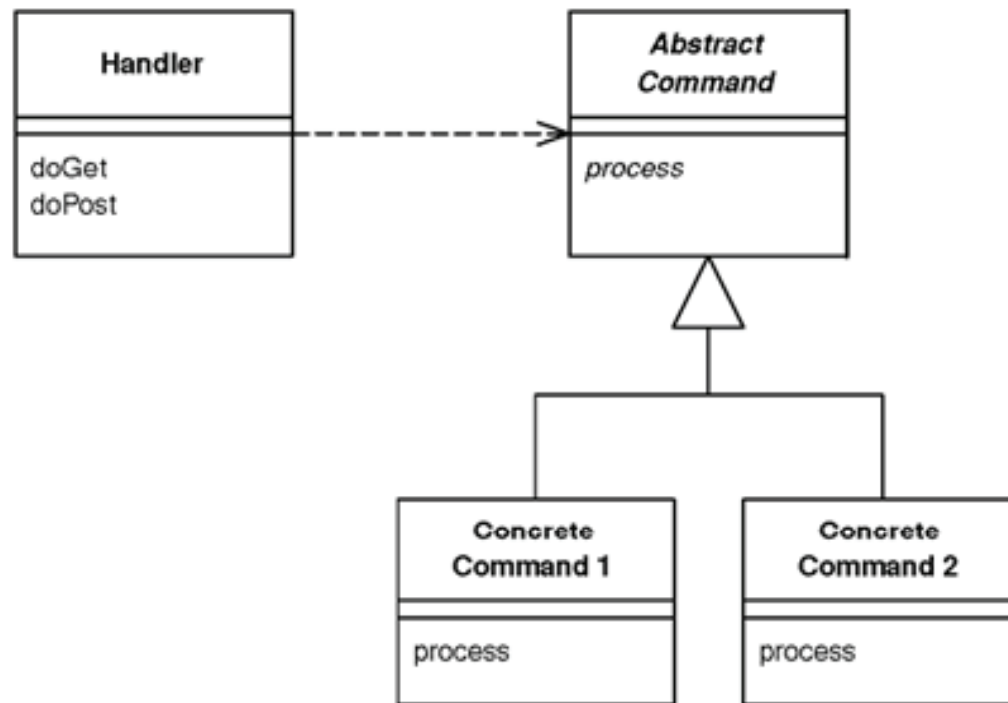
```
class ActionServlet...  
  
    protected void forward(String target,  
                            HttpServletRequest request,  
                            HttpServletResponse response)  
        throws IOException, ServletException  
    {  
        RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(target);  
        dispatcher.forward(request, response);  
    }
```

让视图显示

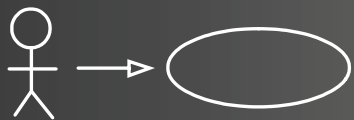


<http://www.umlchina.com>

前端控制器



一个控制器处理站点所有请求



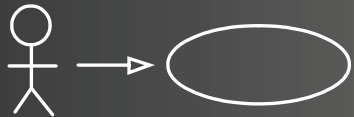
前端控制器

❖ 优点

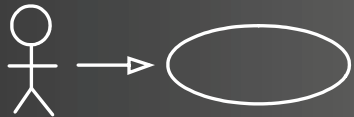
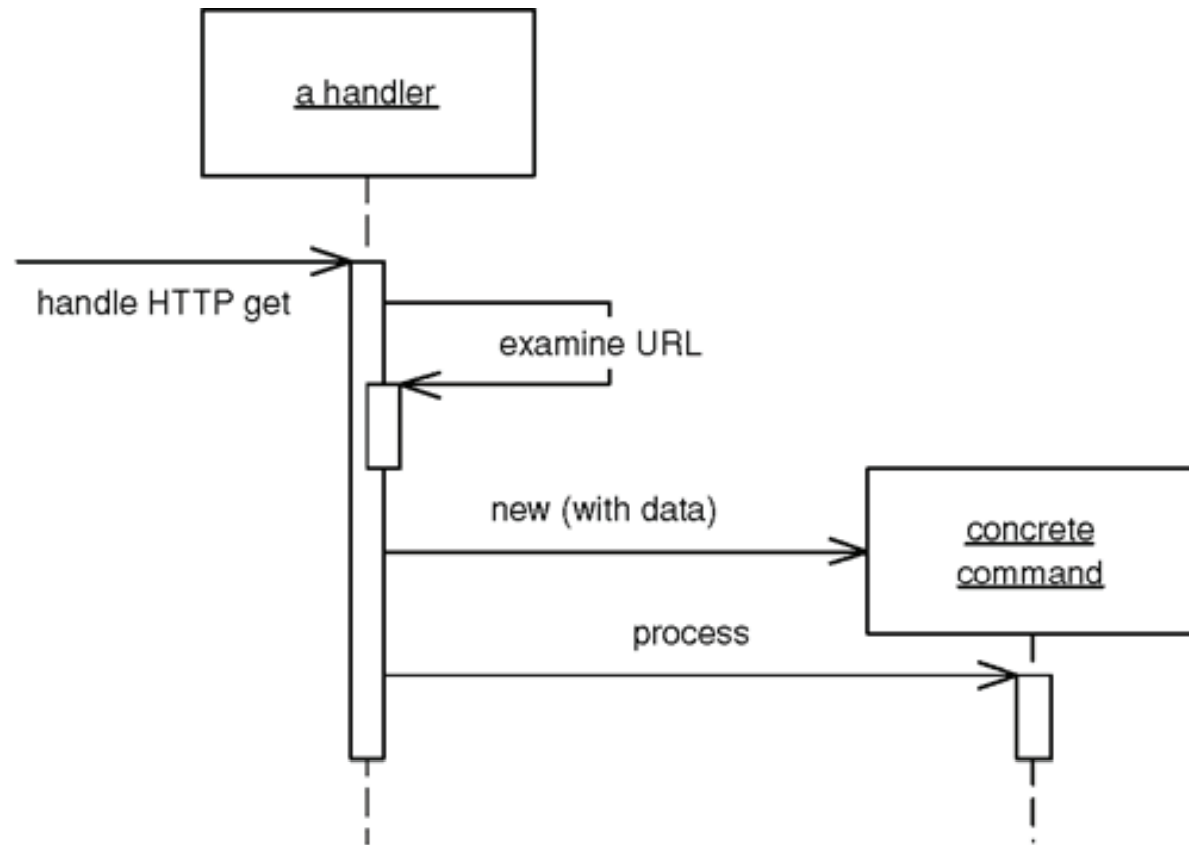
- ❖ 允许提炼重复代码
- ❖ 便于添加装饰动作

❖ 缺点

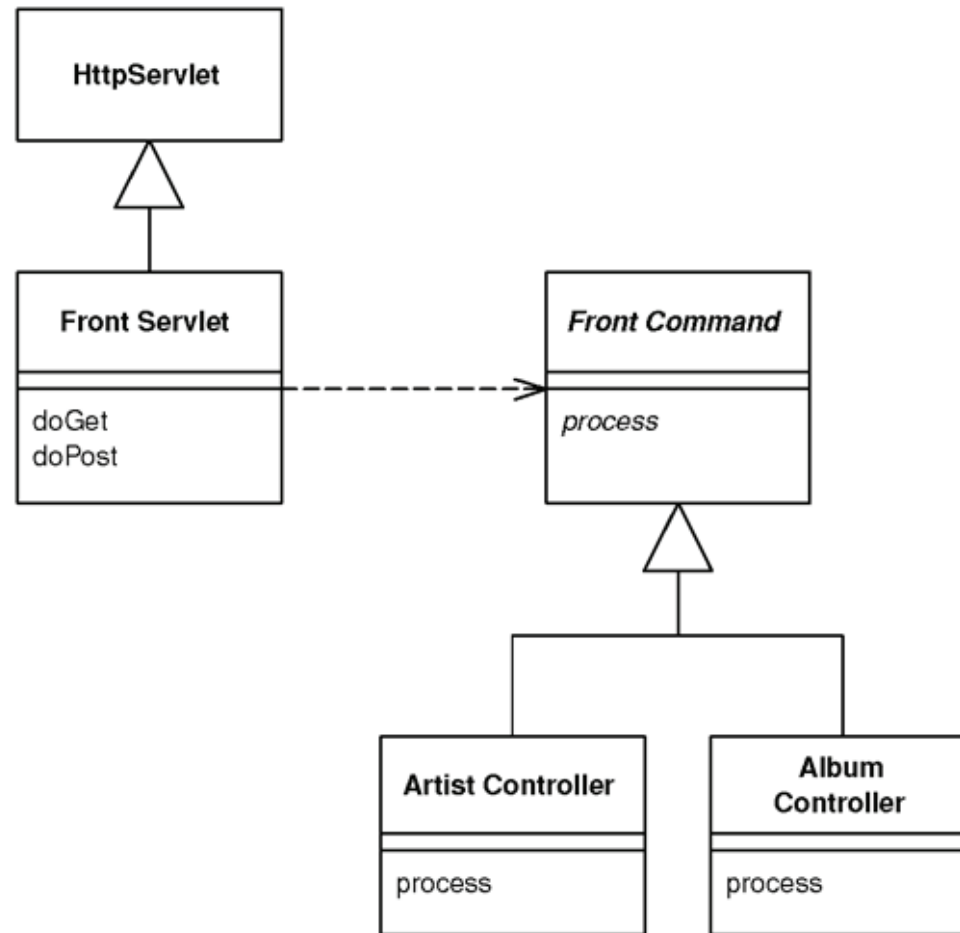
- ❖ 复杂性增加



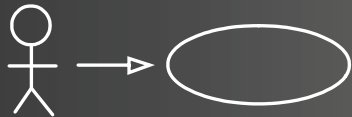
前端控制器交互



前端控制器示例



类图

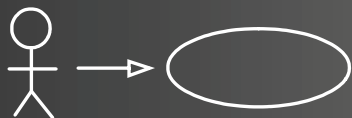


前端控制器示例（续）

```
class FrontServlet...

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        FrontCommand command = getCommand(request);
        command.init(getServletContext(), request, response);
        command.process();
    }
    private FrontCommand getCommand(HttpServletRequest request) {
        try {
            return (FrontCommand) getCommandClass(request).newInstance();
        } catch (Exception e) {
            throw new ApplicationException(e);
        }
    }
    private Class getCommandClass(HttpServletRequest request) {
        Class result;
        final String commandClassName =
            "frontController." + (String) request.getParameter("command") + "Command";
        try {
            result = Class.forName(commandClassName);
        } catch (ClassNotFoundException e) {
            result = UnknownCommand.class;
        }
        return result;
    }
}
```

Handler



<http://www.umlchina.com>

前端控制器示例（续）

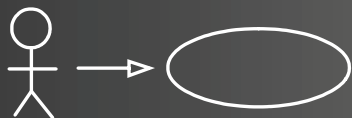
```
class FrontCommand...

    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;
    public void init(ServletContext context,
                    HttpServletRequest request,
                    HttpServletResponse response)
    {
        this.context = context;
        this.request = request;
        this.response = response;
    }
}

class FrontCommand...

    abstract public void process() throws ServletException, IOException ;
    protected void forward(String target) throws ServletException, IOException
    {
        RequestDispatcher dispatcher = context.getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
}
```

抽象Command



前端控制器示例（续）

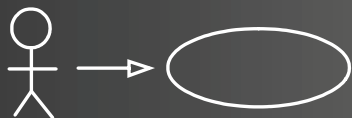
```
class ArtistCommand...

    public void process() throws ServletException, IOException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        request.setAttribute("helper", new ArtistHelper(artist));
        forward("/artist.jsp");
    }

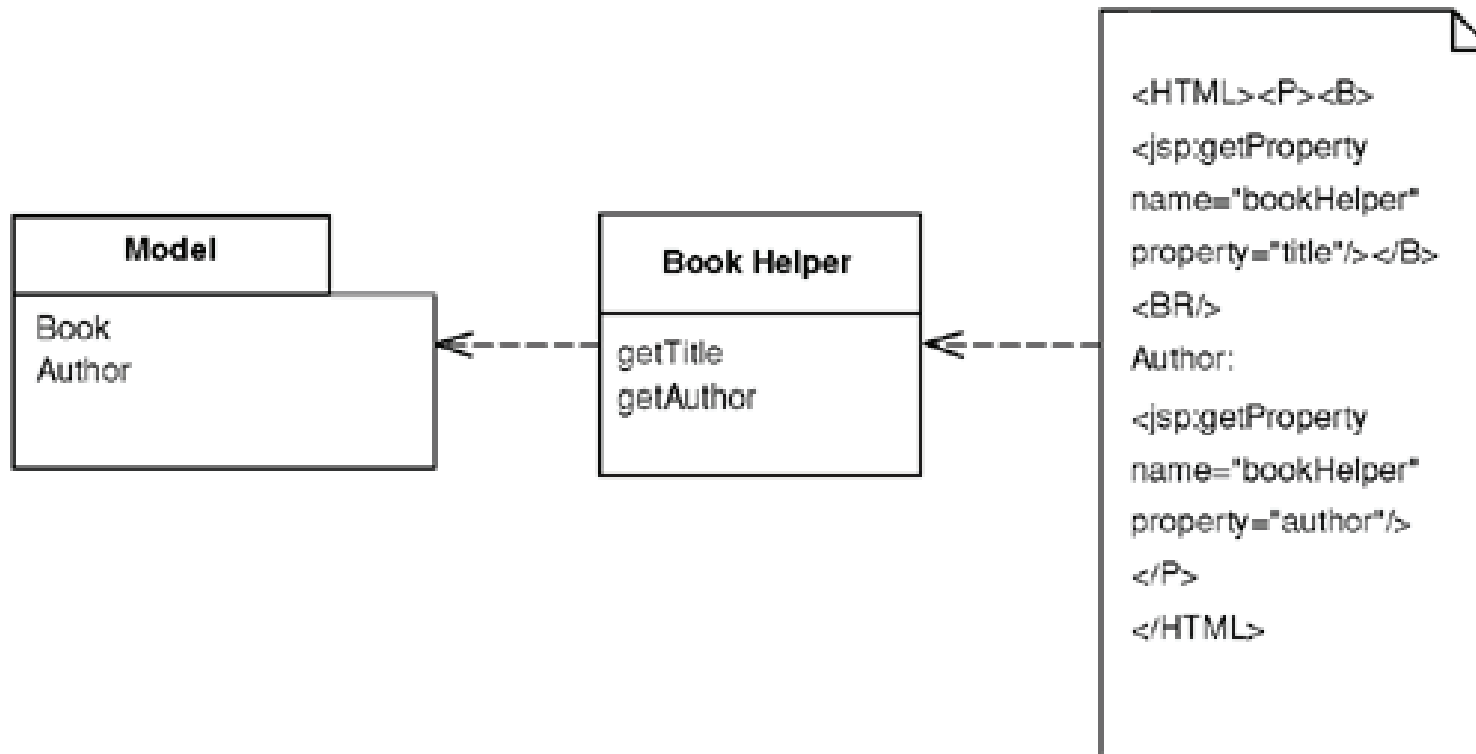
class UnknownCommand...

    public void process() throws ServletException, IOException {
        forward("/unknown.jsp");
    }
```

具体Command



模板视图



在HTML页面嵌入标记



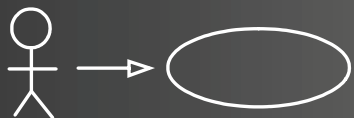
模板视图

❖ 优点

- ❖ 便于组合网页内容
- ❖ 支持美工和程序员分离

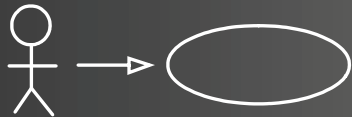
❖ 缺点

- ❖ 网页很容易变乱
- ❖ 难测试

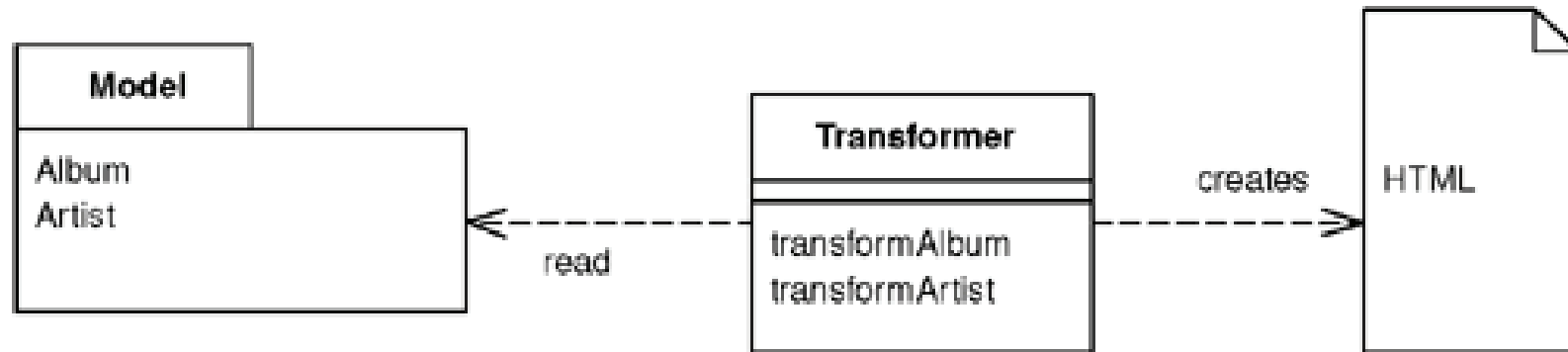


模板视图示例

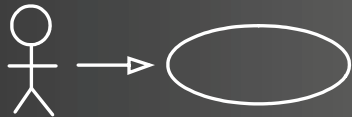
```
<tr class="Title" bgcolor="<%=bgColor %>">  
  <td><a href="AddMember.asp?userid=<%=id  
    %>">Up</a></td>  
  <td><%=login %></td>  
  <td><%=name %></td>  
  <td><%=divisionname %></td>  
  <td><%=email %></td>  
</tr>
```



转换视图



逐项处理领域数据，把它们转成HTML



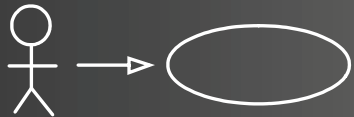
转换视图

❖ 优点

- ❖ 强制分离数据和表现
- ❖ 可维护性
- ❖ 可测试性

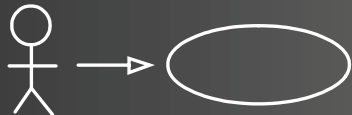
❖ 缺点

- ❖ 对美工不直观
- ❖ 工具支持不如模板视图

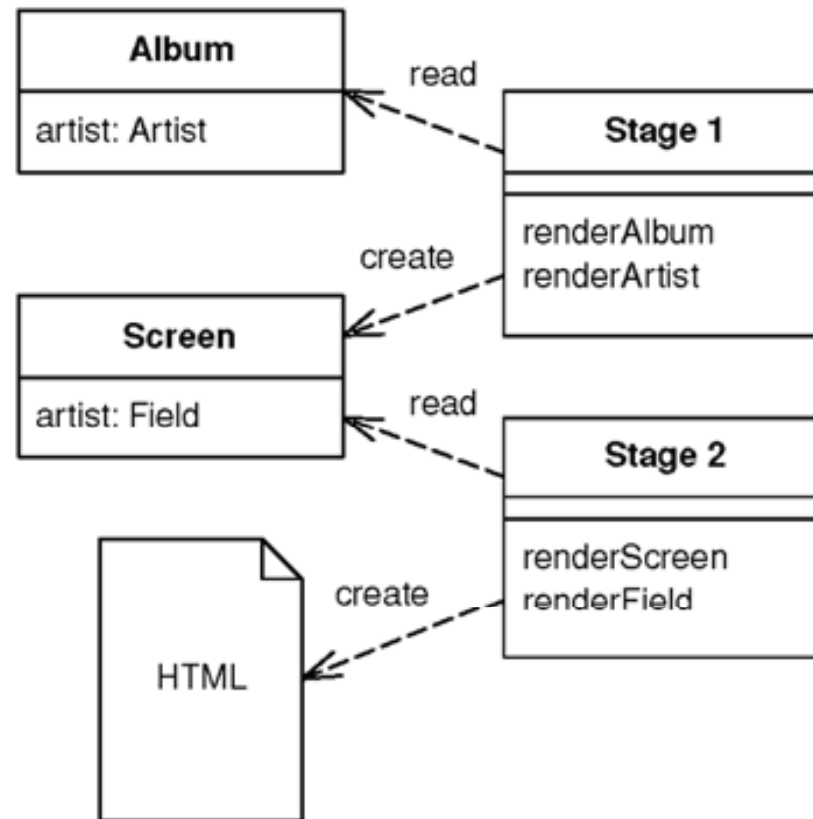


转换视图示例

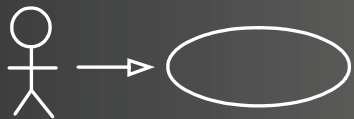
```
<xsl:template match="album">
  <HTML><BODY bgcolor="white">
    <xsl:apply-templates/>
  </BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
  <h1><xsl:apply-templates/></h1>
</xsl:template>
<xsl:template match="artist">
  <P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```



两步视图



用两个步骤来把领域数据转成HTML



两步视图

- ❖ 基于转换视图和模板视图

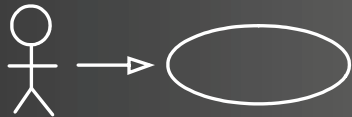
- ❖ 适用：多种外观的应用

- ❖ 优点

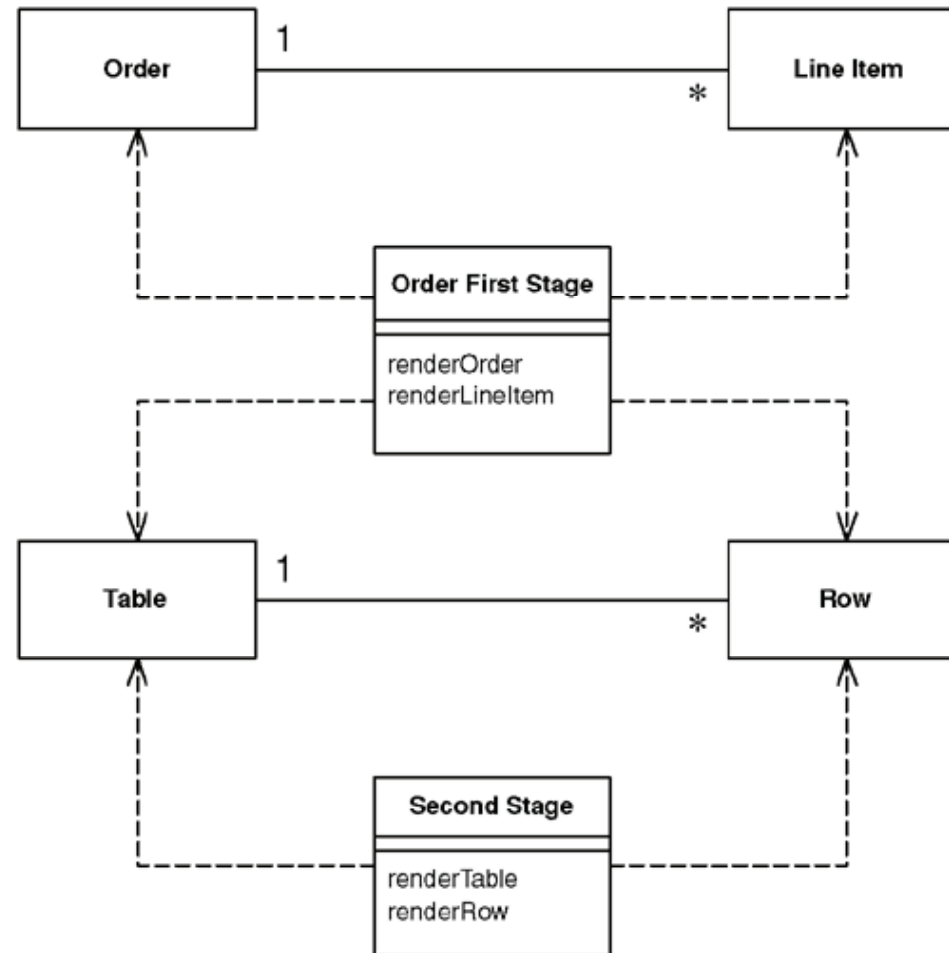
 - ❖ 可维护性

- ❖ 缺点

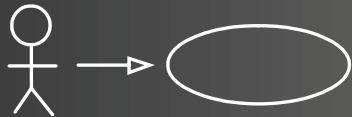
 - ❖ 性能



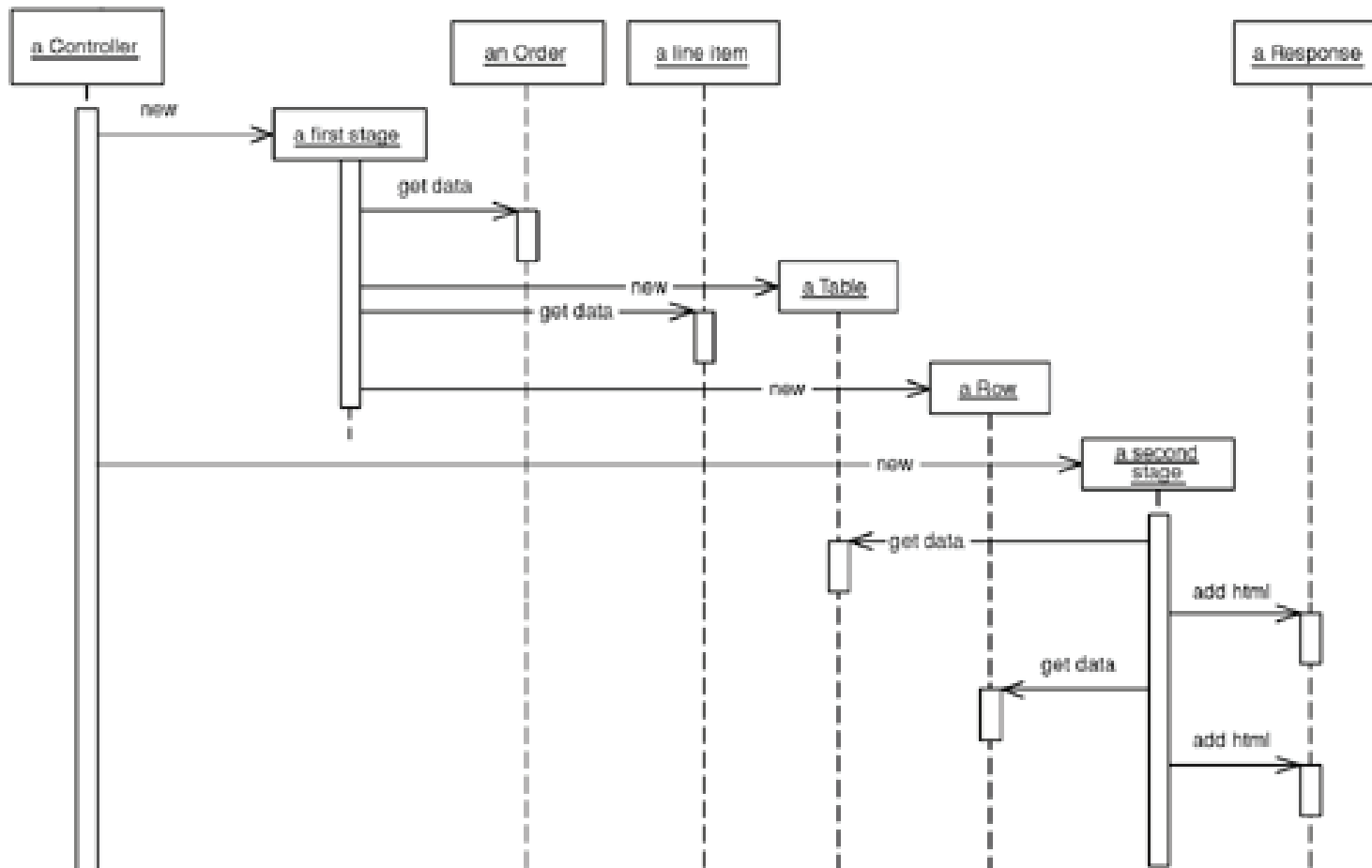
两步视图示例



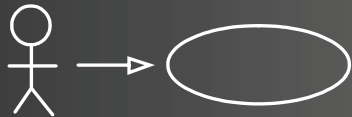
类图



两步视图示例（续）



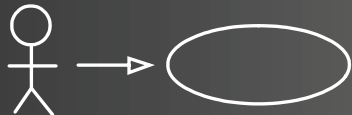
序列图



两步视图示例（续）

```
<%@ taglib uri="2step.tld" prefix = "2step" %>
<%@ page session="false"%>
<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
<2step:screen>
<2step:title><jsp:getProperty name = "helper" property = "title"/></2step:title>
<2step:field label = "Artist"><jsp:getProperty name = "helper" property = "artist"/></
  2step:field>
<2step:table host = "helper" collection = "trackList" columns = "title, time"/>
</2step:screen>
```

第一阶段



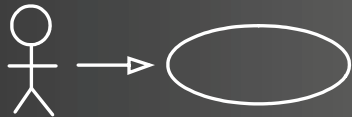
<http://www.umlchina.com>

两步视图示例（续）

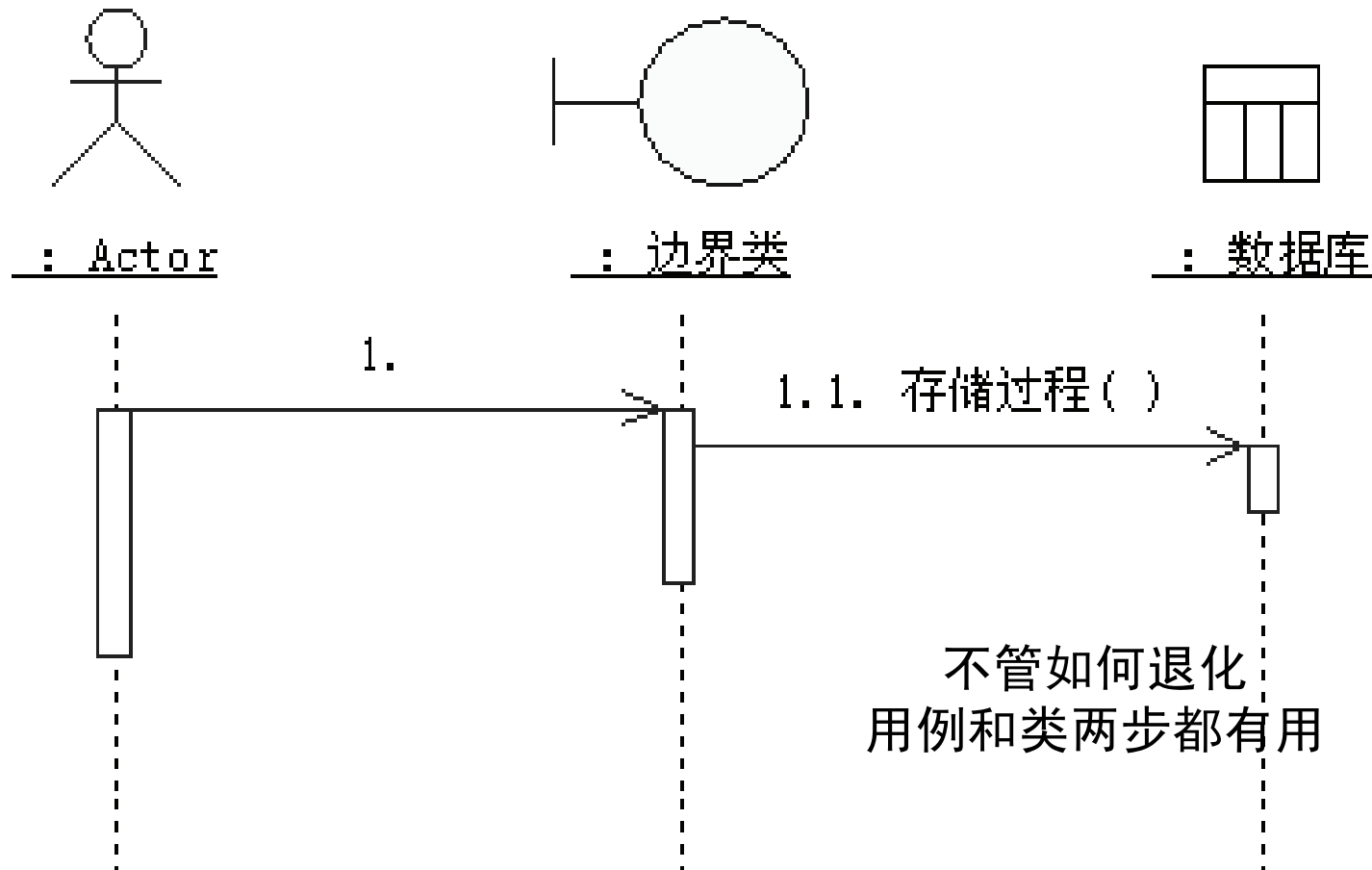
```
class TitleTag...

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("<H1>");
        } catch (IOException e) {
            throw new JspException("unable to print start");
        }
        return EVAL_BODY_INCLUDE;
    }
    public int doEndTag() throws JspException {
        try {
            pageContext.getOut().print("</H1>");
        } catch (IOException e) {
            throw new JspException("unable to print end");
        }
        return EVAL_PAGE;
    }
}
```

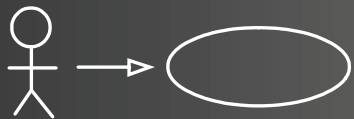
第二阶段



架构的退化

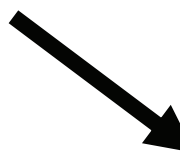


可以退化到存储过程



映射存储过程

```
private void ....  
{  
    联系人 o联系人1, o联系人2;  
  
    ....  
    o联系人1.合并联系人(o联系人2);  
}
```



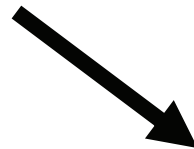
```
create PROCEDURE 合并联系人  
(  
    @联系人1_ID int,  
    @联系人2_ID int  
)  
AS
```

对象→ID



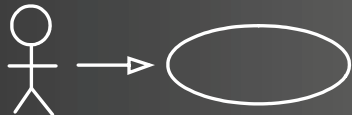
映射存储过程

```
public decimal OrderTotal{  
    get { return _orderTotal; }  
    set { _orderTotal = value; }  
}
```



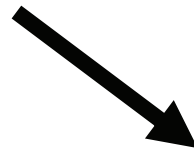
```
select @OrderTotal=OrderTotal from T_Order where ....  
update T_Order set OrderTotal=@OrderTotal  where ....
```

简单属性→直接SQL语句



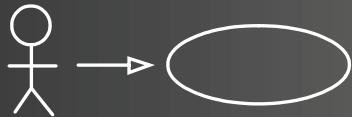
映射存储过程

```
o联系人l=new 联系人(string 姓名,string 地址);
```



```
CREATE PROCEDURE 新增联系人  
(  
    @姓名 nvarchar(500),  
    @地址 nvarchar(500)  
)  
AS
```

对象创建、删除、取集合→简单存储过程

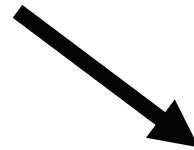


映射存储过程

订单 o.订单;

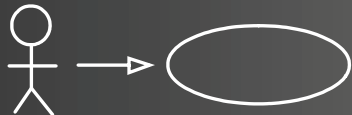
....

供应商 o.供应商=o.订单.寻找最合适供应商();

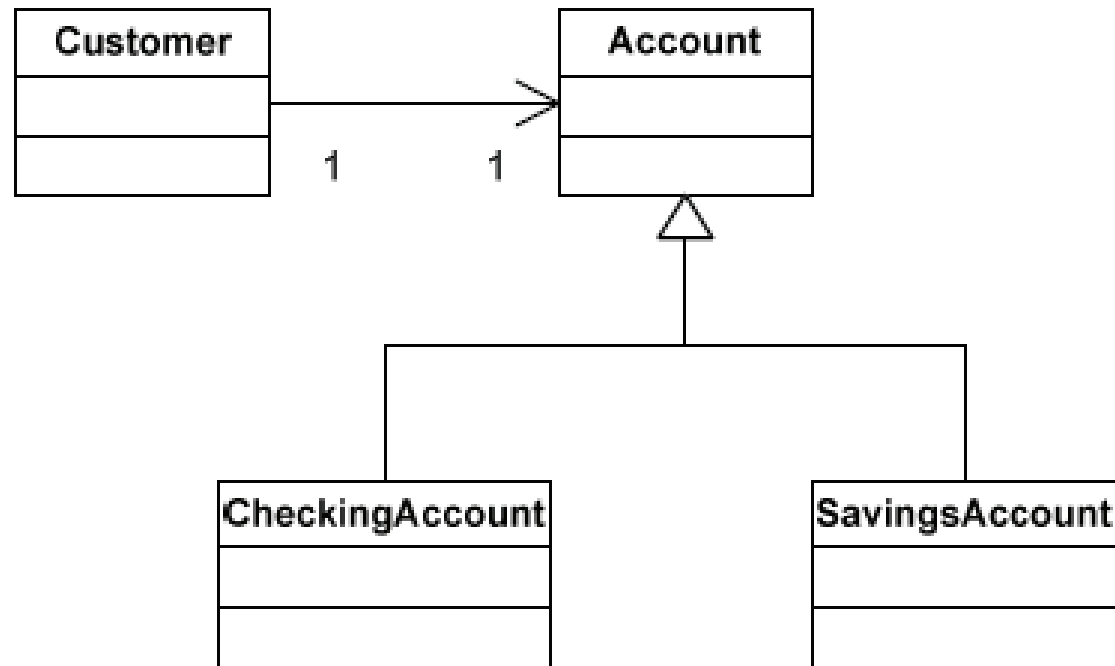


```
CREATE PROCEDURE 寻找最合适供应商  
(  
    @订单_ID int,  
    @供应商_ID int output  
)  
AS
```

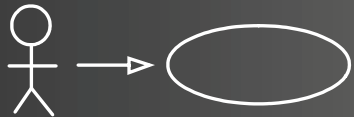
业务职责→复杂存储过程



映射存储过程



多态→类型标志控制 (1)



映射存储过程

id	balance	account_type
1	1000	CheckingAccount
2	1000	SavingsAccount

Account Table

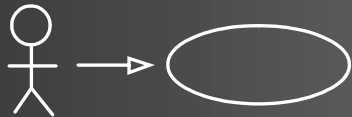
id	account_id	minimum_balance
1	1	900

CheckingAccount Table

id	account_id	amount_interest_paid	date_interest_paid
1	2	10	2003-03-31 00:00:00.000

SavingsAccount Table

多态→类型标志控制 (2)



映射存储过程

```
CREATE PROCEDURE selCheckingAccount (@iID int) AS

SELECT
Account.account_type, Account.id, Account.balance,
CheckingAccount.minimum_balance
FROM
Account, CheckingAccount
WHERE
account_id = @iID AND Account.[id] = @iID

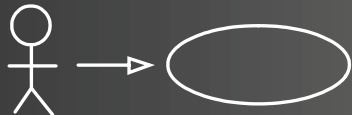
GO

CREATE PROCEDURE selSavingsAccount (@iID int) AS

SELECT
Account.account_type, Account.id, Account.balance,
SavingsAccount.amount_interest_paid, SavingsAccount.date_interest_paid
FROM
Account, SavingsAccount
WHERE
account_id = @iID AND Account.[id] = @iID

GO
```

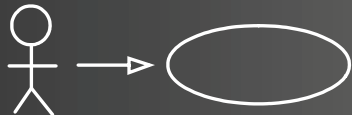
多态→类型标志控制（3）



映射存储过程

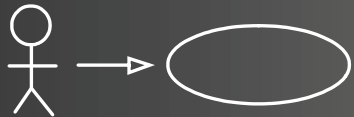
```
CREATE PROCEDURE selAccount (@iID int) AS  
  
DECLARE @sAccountType AS varchar(256)  
  
SELECT  
@sAccountType = account_type  
FROM  
Account  
WHERE  
[id] = @iID  
  
DECLARE @sStoredProc varchar(270)  
  
SET @sStoredProc = 'sel' + @sAccountType  
|  
exec @sStoredProc @iID  
GO
```

多态→类型标志控制 (4)



映射C语言

- ❖ 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

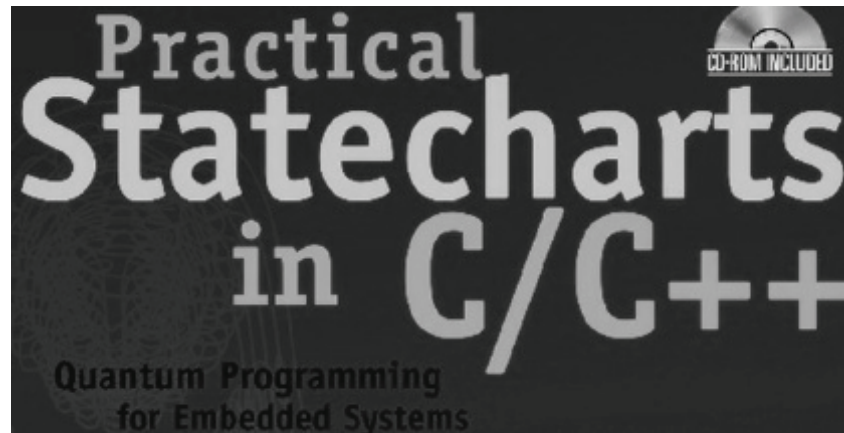


C实现

Object Oriented Programming in C

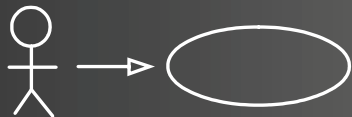
Laurent Deniau

<http://ldeniau.home.cern.ch/ldeniau/html/oopc/oopc.html>



<http://www.quantum-leaps.com/index.htm>

面向对象设计的C语言实现



<http://www.umlchina.com>

C实现

```
/* machine.h */
#ifndef MACHINE_H
#define MACHINE_H
#include <oo.h>

#undef OBJECT
#define OBJECT machine

/* Object interface */
BASEOBJECT_INTERFACE
char const* private(name);
char const* private(year);
float private(height);
float private(price);
BASEOBJECT_METHODS
float constMethod(computeSalePrice);
float constMethod(inquireHeight);
void constMethod(printName);
ENDOF_INTERFACE

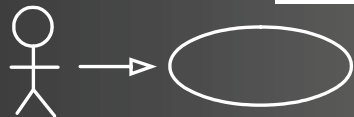
/* Class interface */
CLASS_INTERFACE
t_machine*const classMethod_(new)
    char const name[], char const year[], float height, float price __;
void method_(init) char const name[], char const year[],
float height, float price __;
void method_(copy) t_machine const*const pObj __;
ENDOF_INTERFACE
#endif
```

类名

属性

方法

类函数



C实现

```
/* machine.c */
#include <stdio.h>
#define IMPLEMENTATION
#include <machine.h>

/* Object implementation */
float
constMethodDecl (computeSalePrice)
{ return (this->m.price) * 0.8; }

float
constMethodDecl (inquireHeight)
{ return this->m.height; }

void
constMethodDecl (printName)
{ printf("品牌:\t%s\n", this->m.name); }

BASEOBJECT_IMPLEMENTATION
    methodName (computeSalePrice),
    methodName (inquireHeight),
    methodName (printName)
ENDOF_IMPLEMENTATION
```

```
/* Class implementation */
initClassDecl () {}
dtorDecl ()
{ free((void*)this->m.name);
  this->m.name = NULL;
  free((void*)this->m.year);
  this->m.year = NULL;
}

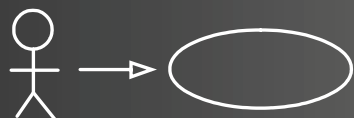
t_machine
classMethodDecl_(*const new) char const name[], char const year[],
float height, float price __
{ t_machine *const this = machine.alloc();
  if (this) machine.init(this, name, year, height, price);
  return this;
}

void
methodDecl_ (init) char const name[], char const year[],
float height, float price __
{ this->m.name = strdup(name);
  this->m.year = strdup(year);
  this->m.height = height;
  this->m.price = price;
}

void
methodDecl_ (copy) t_machine const*const pobj __
{ machine._machine(this);
  machine.init(this, pobj->m.name, pobj->m.year, pobj->m.height,
pobj->m.price);
}

CLASS_IMPLEMENTATION
    methodName (new),
    methodName (init),
    methodName (copy)
ENDOF_IMPLEMENTATION
```

实现



C实现

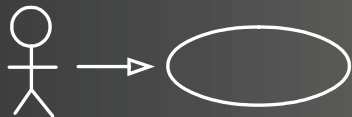
```
/* xmain.c */
#include <stdio.h>
#include <machine.h>
```

产生machine对象，并传递消息

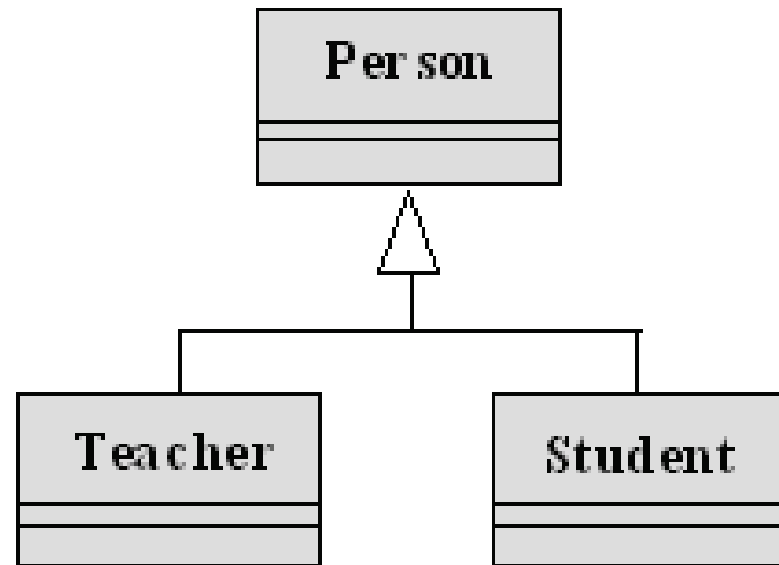
```
int main(void)
{
    float sale_price, height;
    t_machine *a = machine.new("索尼", "2007", 0.83, 5000.0);
    /* send messages */
    sendMsg(a, printName);
    sale_price = sendMsg(a, computeSalePrice);
    height = sendMsg(a, inquireHeight);
    printf("height : %.2f\n", height);
    printf("sale_price : %.2f\n", sale_price);

    /* delete object */
    delete(a);

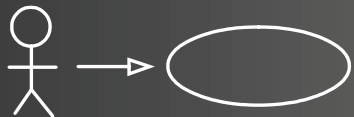
    getch();
    return EXIT_SUCCESS;
}
```



C实现



泛化



C实现

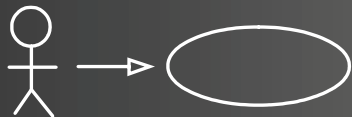
```
/* person.h */
#ifndef PERSON_H
#define PERSON_H

#include <oooc.h>
#undef OBJECT
#define OBJECT Person

/* Object interface */
BASEOBJECT_INTERFACE
    char *private(name);
    int private(age);

BASEOBJECT_METHODS
    void constMethod(Display);
    void constMethod(YearOfBirth);
ENDOF_INTERFACE

/* Class interface */
CLASS_INTERFACE
    t_Person *const classMethod_(new) char const name[], int age __;
    void method_(init) char const name[], int age __;
- ENDOF_INTERFACE
#endif
```



C实现

```
≡ /* person.c */
#include <stdio.h>
#define IMPLEMENTATION
#include <person.h>

/* Object implementation */
≡ void constMethodDecl (Display)
{ printf("Name: %s, Age: %d\n", this->m.name, this->m.age); }

≡ void constMethodDecl (YearOfBirth)
{ printf("YearOfBirth: %d\n", 2006 - this->m.age); }

≡ BASEOBJECT_IMPLEMENTATION
method_name (Display),
method_name (YearOfBirth)
ENDOF_IMPLEMENTATION

/* Class implementation */
≡ initClassDecl ()
{ } /* required */
≡ dtorDecl () /* required */
{ free((void*)this->m.name);
  this->m.name = NULL;
}

t_Person classMethodDecl_ (*const new) char const name[], int age __
{ t_Person *const this = Person.alloc();
  if (this) Person.init(this, name, age);
  return this;
}

void methodDecl_(init) char const name[], int age __
{ this->m.name = strdup(name);
  this->m.age = age;
}

CLASS_IMPLEMENTATION
method_name (new),
method_name (init)
ENDOF_IMPLEMENTATION
```



C实现

```
/* teacher.h */
#ifndef TEACHER_H
#define TEACHER_H
#include <person.h>

#undef OBJECT
#define OBJECT Teacher

/* Object interface */
OBJECT_INTERFACE
INHERIT_MEMBERS_OF(Person);
OBJECT_METHODS
INHERIT_METHODS_OF(Person);
ENDOF_INTERFACE

/* Class interface */
CLASS_INTERFACE
t_Person * const classMethod_(new) char const name[], int age __;
void method_(init) char const name[], int age __;
ENDOF_INTERFACE
#endif
```

表明是Person子类

```
/* teacher.c */
#include <stdio.h>
#define IMPLEMENTATION
#include <teacher.h>

/* Object implementation */
OBJECT_IMPLEMENTATION
SUPERCLASS(Person)
ENDOF_IMPLEMENTATION

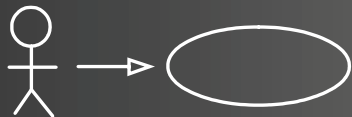
/* Class implementation */
initClassDecl() /* required */
{ initSuper(Person); }

dtorDecl() /* required */
{ Person._Person( super(this, Person) ); }

t_Teacher classMethodDecl_(*const new) char const name[], int age __
{ t_Teacher *const this = Teacher.alloc();
  if (this) Teacher.init(this, name, age);
  return this;
}

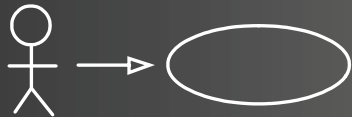
void methodDecl_(init) char const name[], int age __
{ Person.init( super(this, Person), name, age); }

CLASS_IMPLEMENTATION
methodName(new),
methodName(init)
ENDOF_IMPLEMENTATION
```



其他图

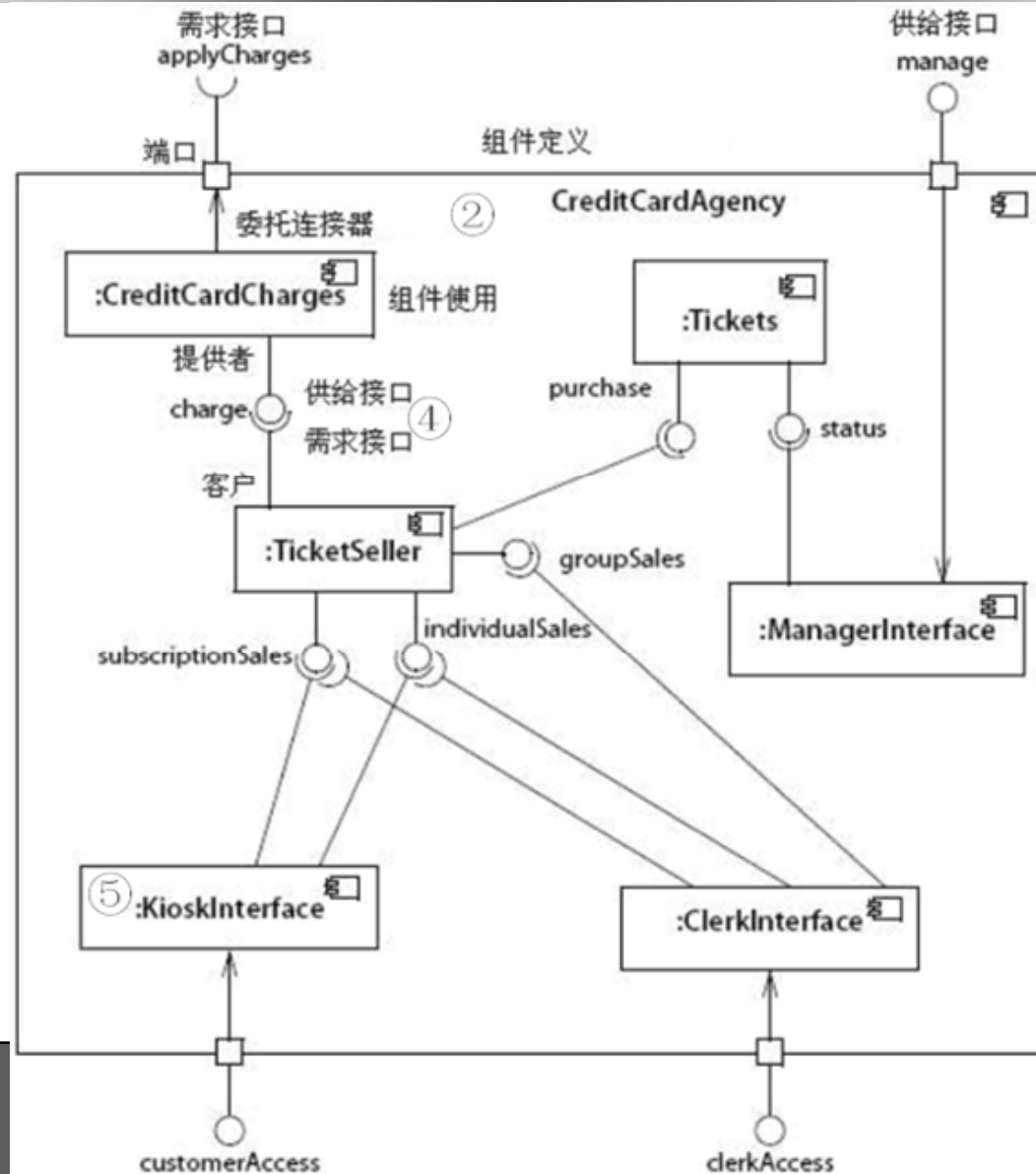
- ❖ 构件图（Component Diagram）— 类的逻辑分包
- ❖ 部署图（Deployment Diagram）— 构件的物理分布
- ❖ 包图 — 各种元素分组



A diagram on a dark gray background. On the left is a white stick figure. An arrow points from the stick figure to a white oval on the right.



构件图: UML2.x



构件图

① 《database》不见了

UML1: 构件可以用来表示物理结构, 象数据库、DLL、EXE、JSP...等

UML2: 由部署图中的工件承担。构件的重点已经从UML1中的物理视图转向了更加逻辑的概念, 这样它们就能够在概念模型中使用。

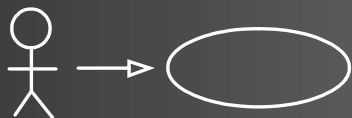
② 构件分成了构件定义和构件使用两部分来描述, 把内和外分开了。

③ 四个执行者没有了, 变成了四个端口 (Port)

和Supervisor执行者对应的是manage端口, 和Customer执行者对应的是customerAccess端口。

构件只定义自己的“协议”, 不把执行者拉进来是合适的。

端口是UML2中出现的新概念, 它可以看作是一些经常一起使用的供给接口和需求接口的组织 (可以对照PC机上的硬件端口来理解)。



构件图

④接口→需求接口和供给接口

需求接口：构件需要的服务

供给接口：构件提供的服务

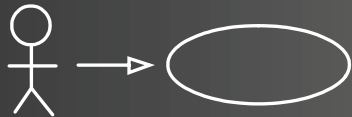
如果一个构件有一个供给接口，另一个构件有相同的需求接口，这两个构件就可以“无缝集成”，无需另外的结构。

基于构件的开发（CBD）。构件代表着一个独立开发、独立购买的，可以置换和按需要组装的单元，就像人们到五金店买零件组装一样。UML2更强调这个特点，并不在意构件是用面向对象方法或者面向过程方法构造的，甚至，不在意它是软件还是硬件。

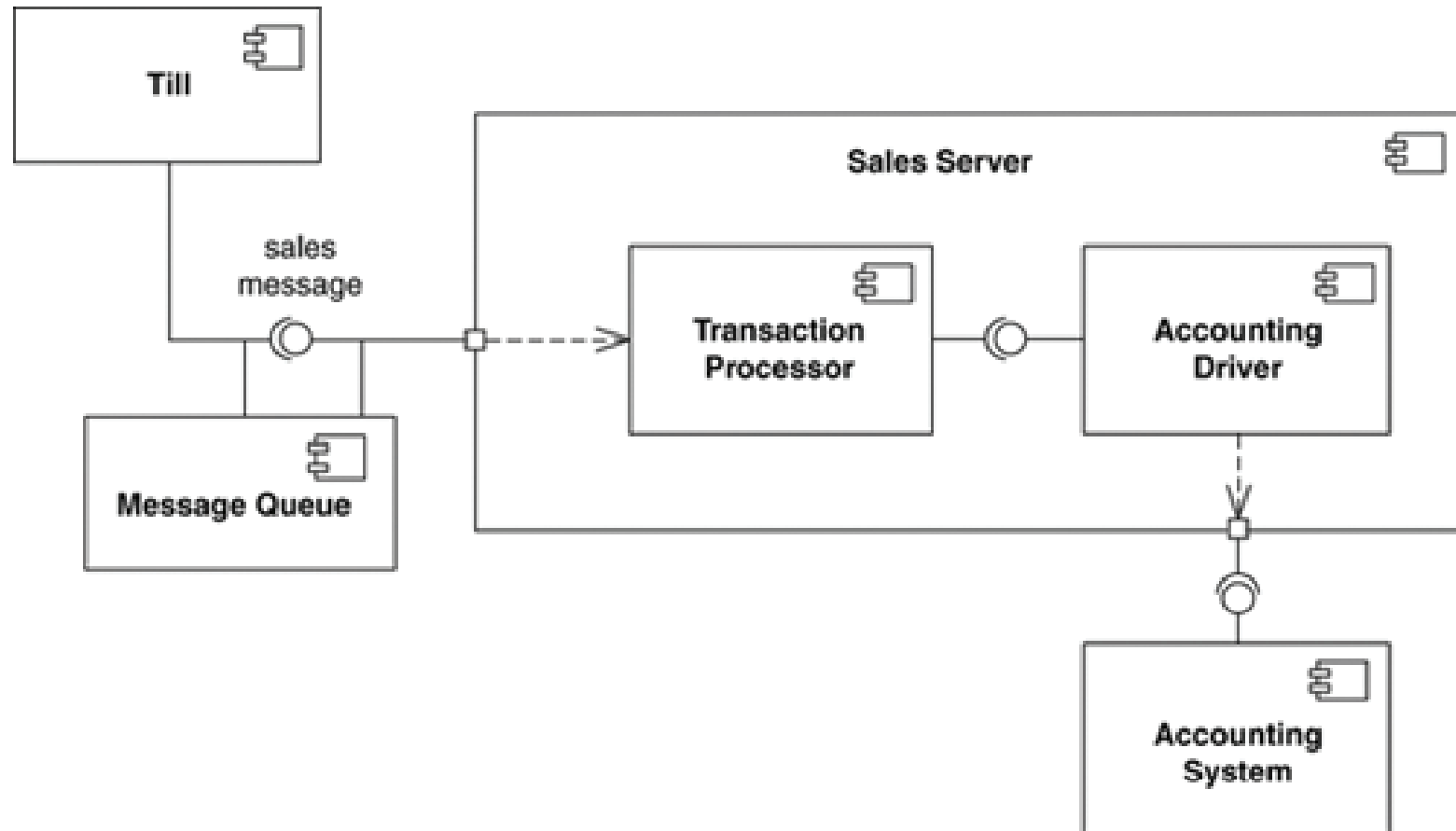
⑤KioskInterface前面有个冒号

“KioskInterface”也是一个类。

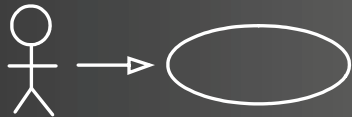
UML2：构件图和普通类图之间没有严格的界限，也不再有单独的一个图标来表示构件。KioskInterface的框框就是一个类框，构件的标志缩成一个小图标放在右上角，也可以用构造型《component》取代图标。



构件图



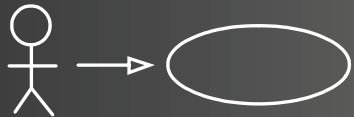
构件的装配



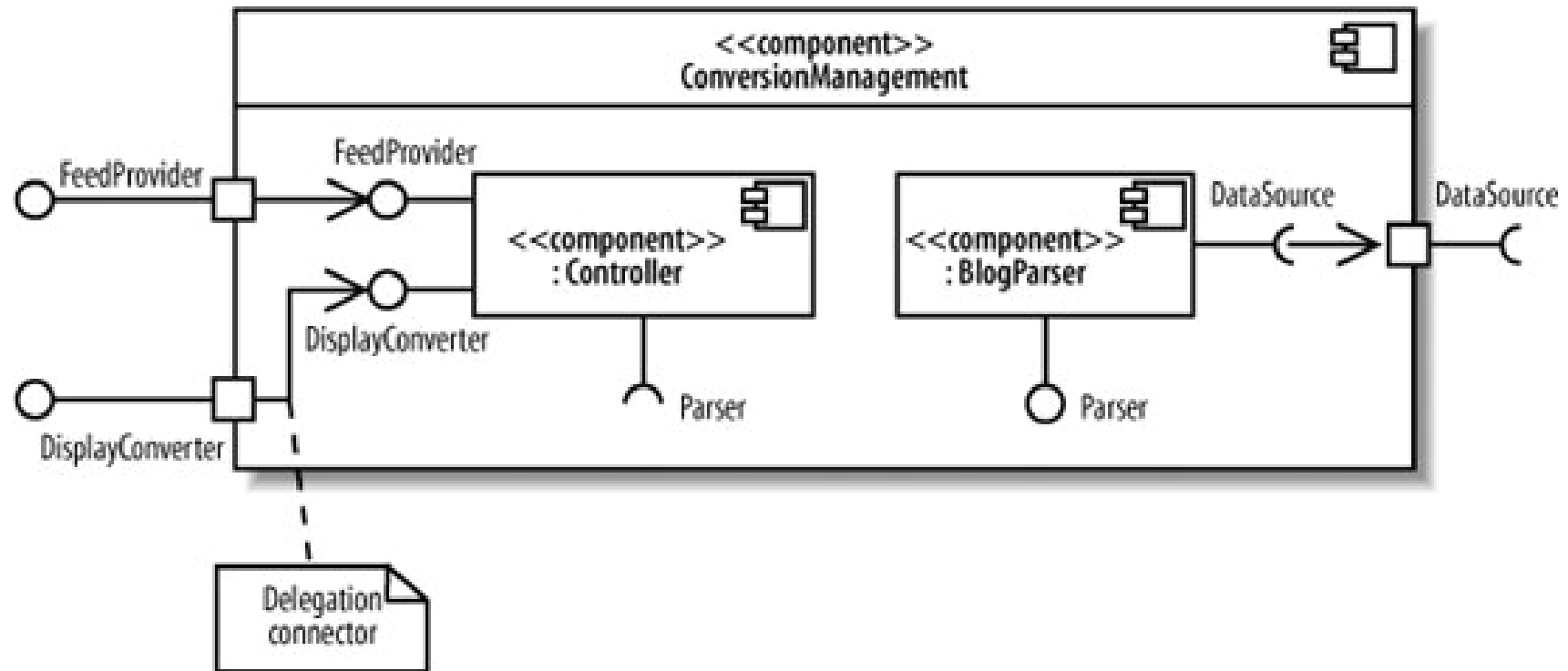
构件图



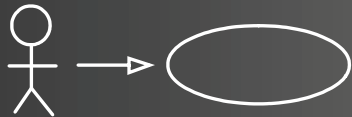
构件的装配



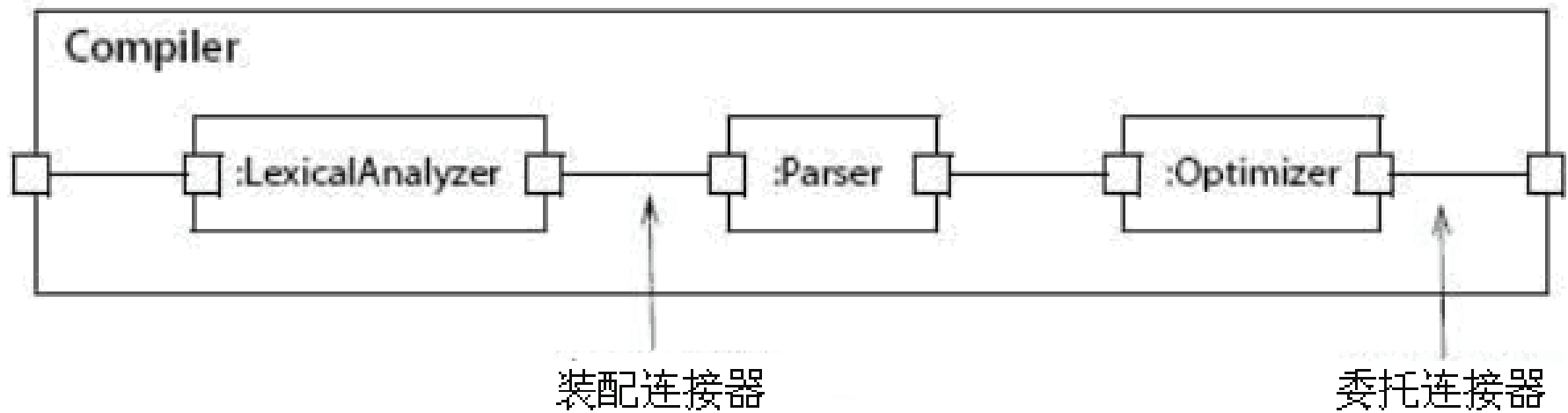
构件图



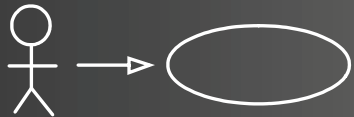
连接器一端口之间的桥梁



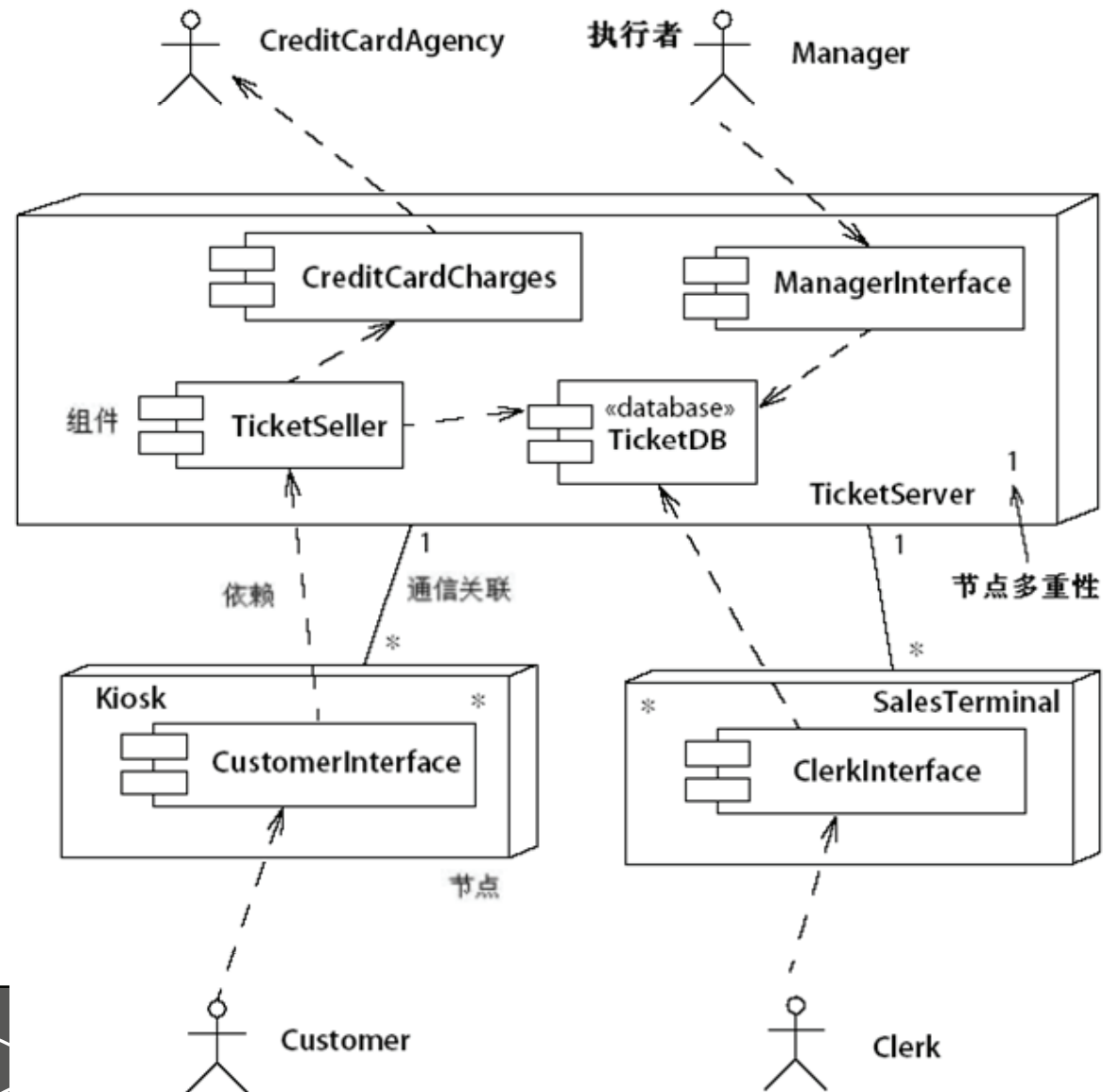
构件图



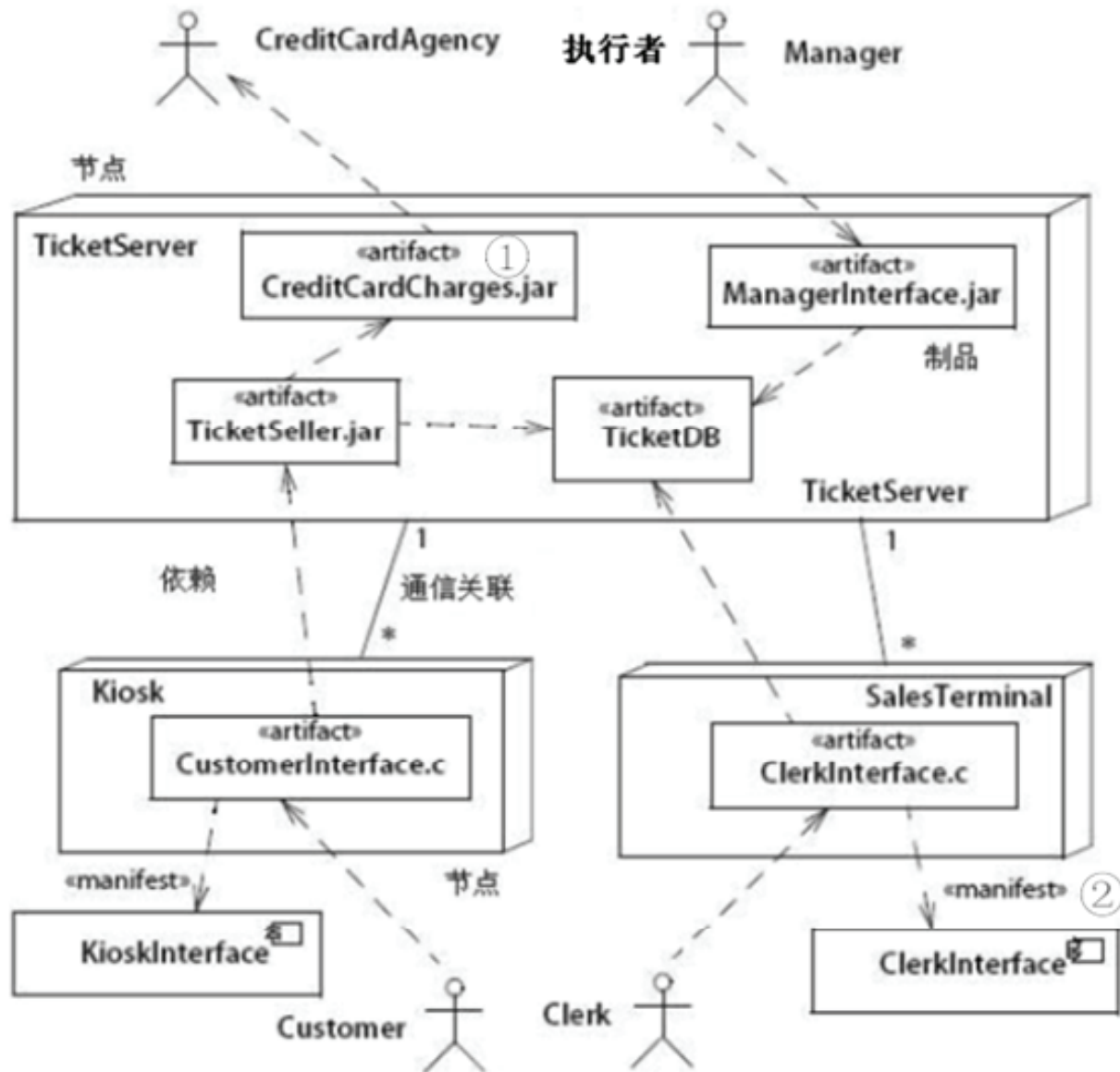
两种连接器



部署图: UML1.x



部署图: UML2.x



部署图

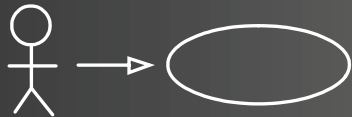
① 《artifact》的构造型

部署在UML2中被重新定位，被应用于工件而不是模型元素。UML1构件图中的“TicketDB”，作为一个工件在这里出现了。另外，上面UML1构件图中被取消的参与者，也在部署图中出现了。

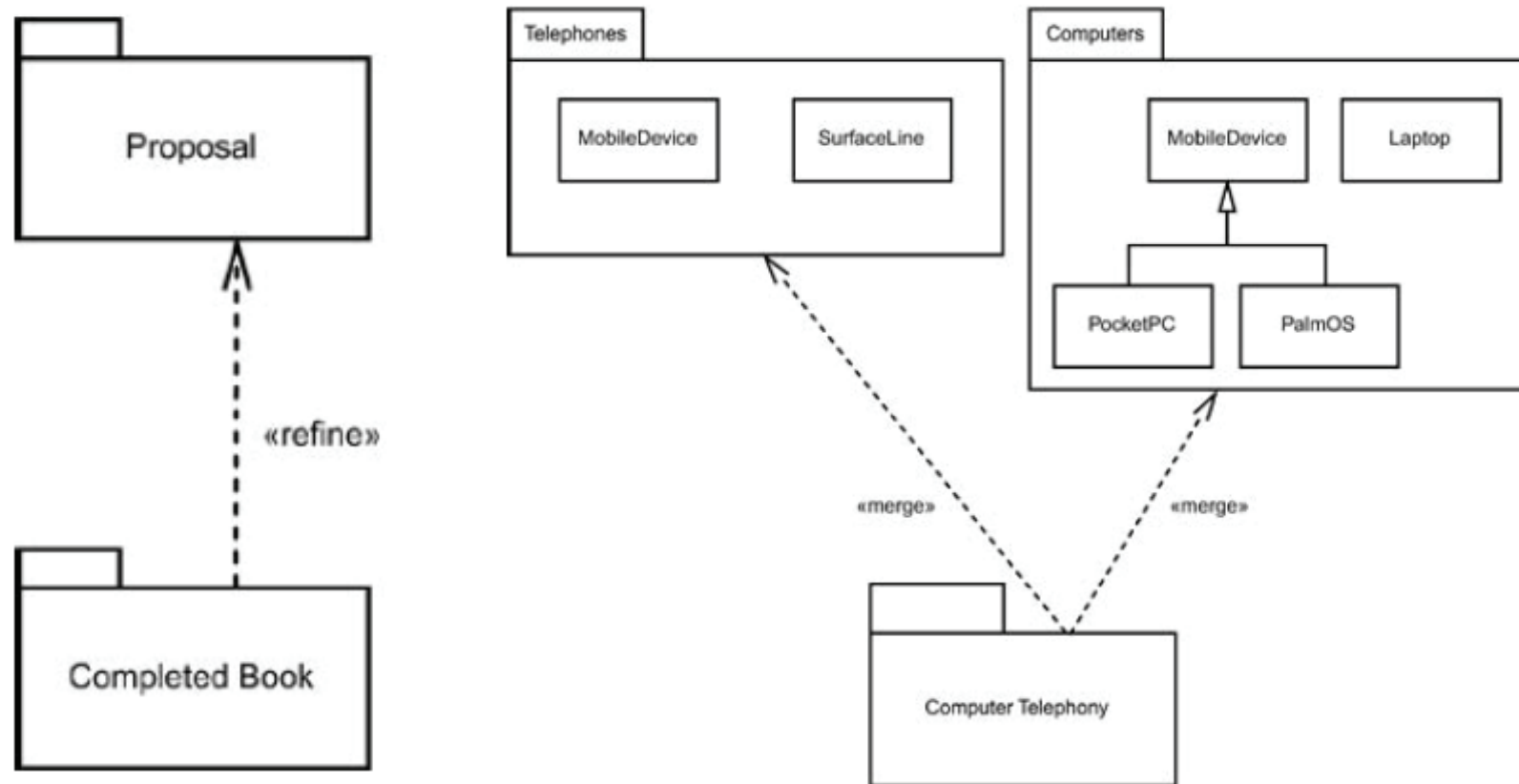
② 构件和工件之间的《manifest》关系

UML2：模型元素和工件（artifact）分开了，而追踪模型元素与实现它们的工件之间的对应关系非常重要，把它们联系起来的就是显现（manifestation）。

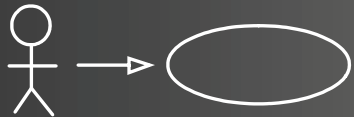
显现：将一个模型元素物理实现为一个工件。在软件中，模型最终会被实现为一组工件，诸如不同类型的一些文件。工件是被部署到物理节点（node）上的实体，物理节点可以是计算机或存储设备等。



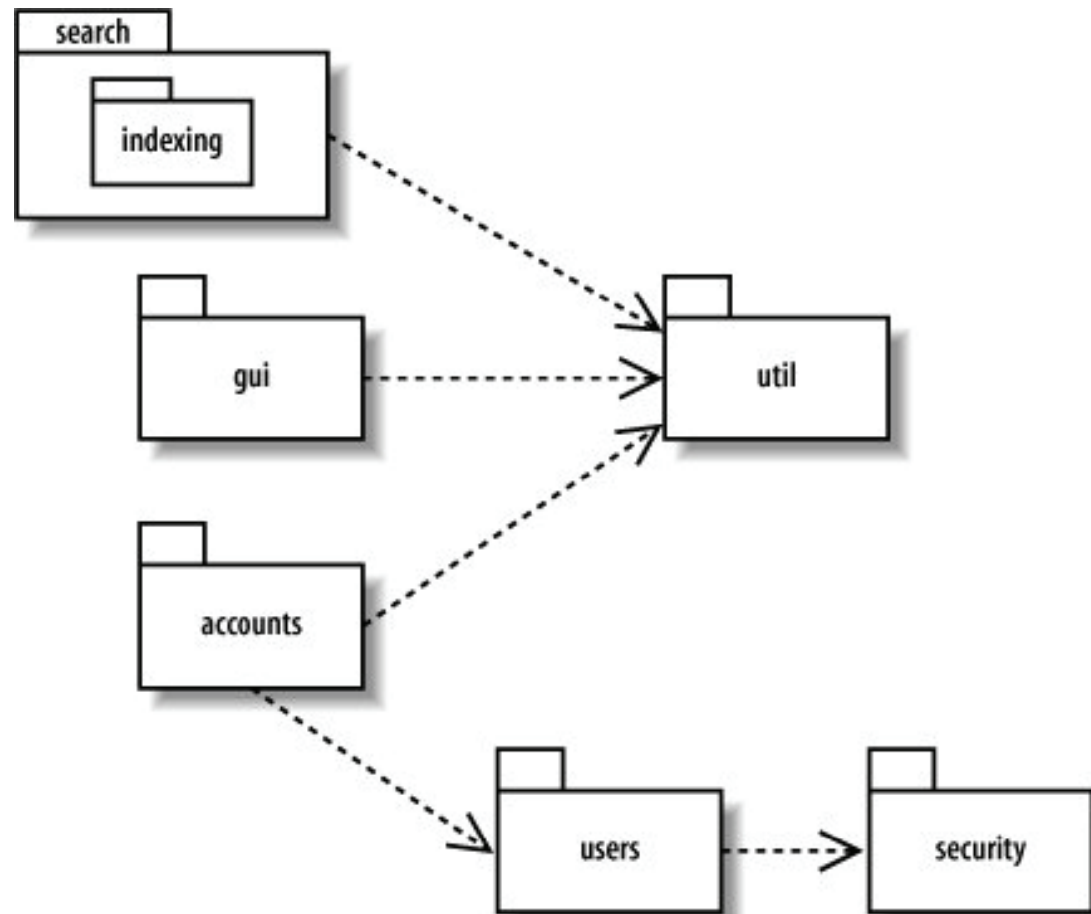
包图



组织各种UML元素—正式成为一种图



包图



依赖

