第4章 面向对象的软件分析与设计

4.1 4+1模型及UML语言实现
4.2 面向对象的软件工程
4.3 用例图
4.4 活动图
4.5 用户界面设计
4.6 类图
4.7 交互图
4.8 包图
4.9 系统与子系统
4.10 部署图

第4章 面向对象的软件分析与设计

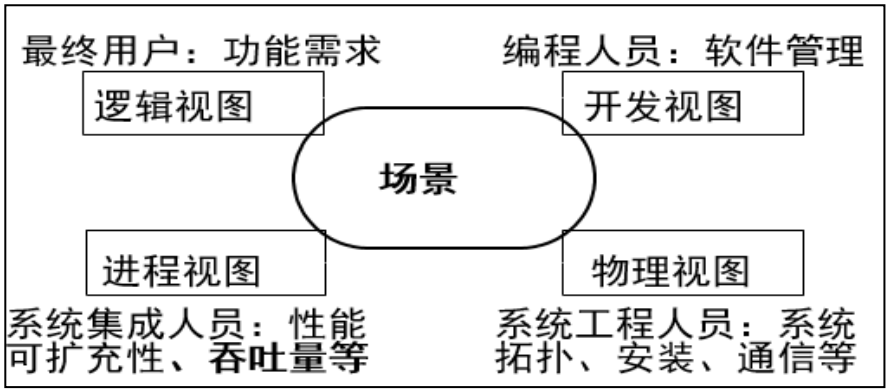
4.6类图

4.6.1面向对象软件分析与设计
4.6.2 类(Class)
4.6.3 接口(Interface)
4.6.4关系(relationship)
4.6.5 领域建模（类图）

第4章 面向对象的软件分析与设计

4.6.1面向对象软件分析与设计

- 4+1模型的UML语言实现



用例图、活动图、顺序图、ER图、系统界面、类图、包图、软件目录结构、部署图

- 基于用例驱动的面向对象软件分析与设计

软件分析：用例图、活动图、ER图、系统界面

软件设计：类图、顺序图、包图、软件目录结构、部署图

4.6类图

4.6.2 类(Class)

- 概念

具有相同属性(attributes)、操作(operations)、关系(relationships)和语义(semantics)的对象的抽象。

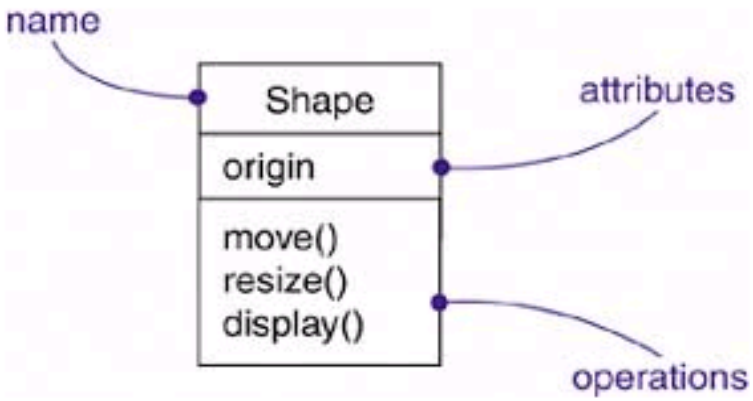
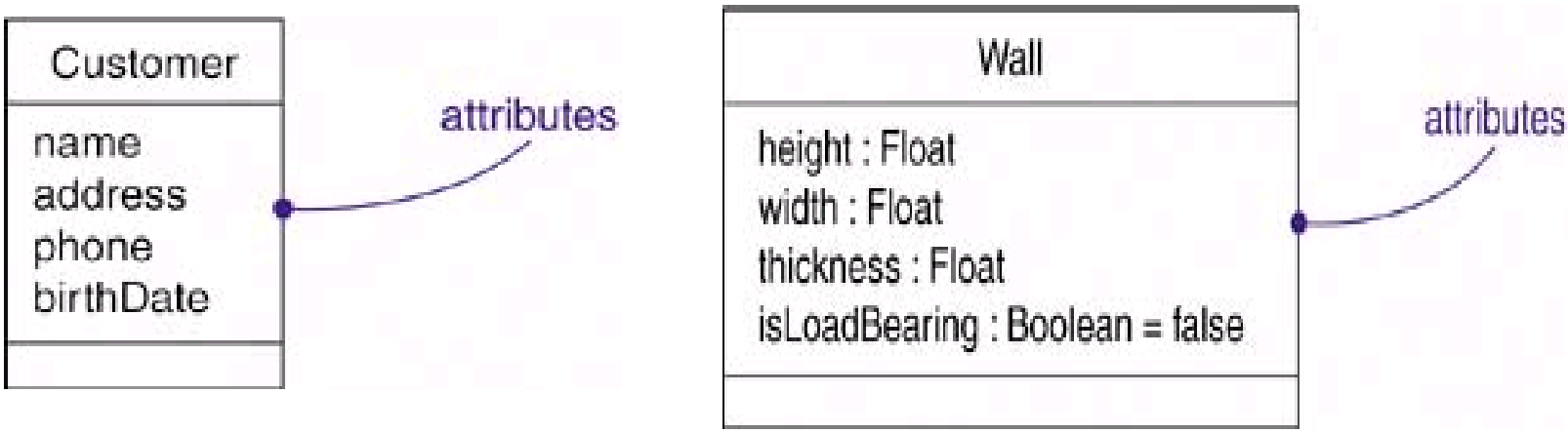


Figure . Classes

4.6.2 类(Class)

- 属性(Attributes)



4.6.2 类(Class)

- 操作(Operations)

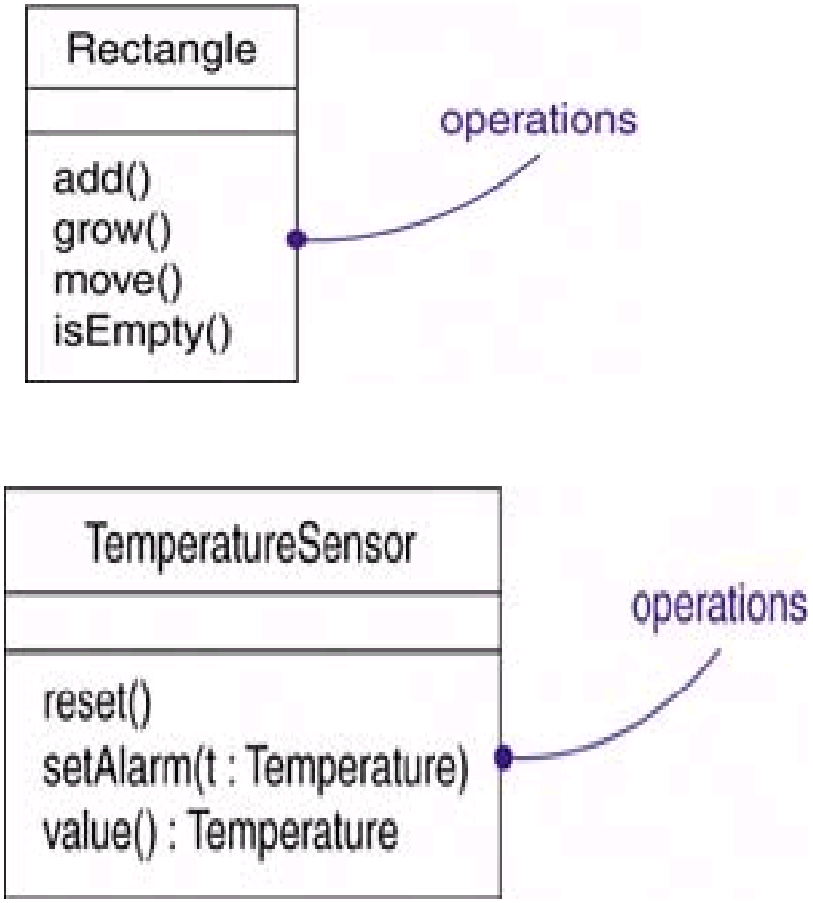


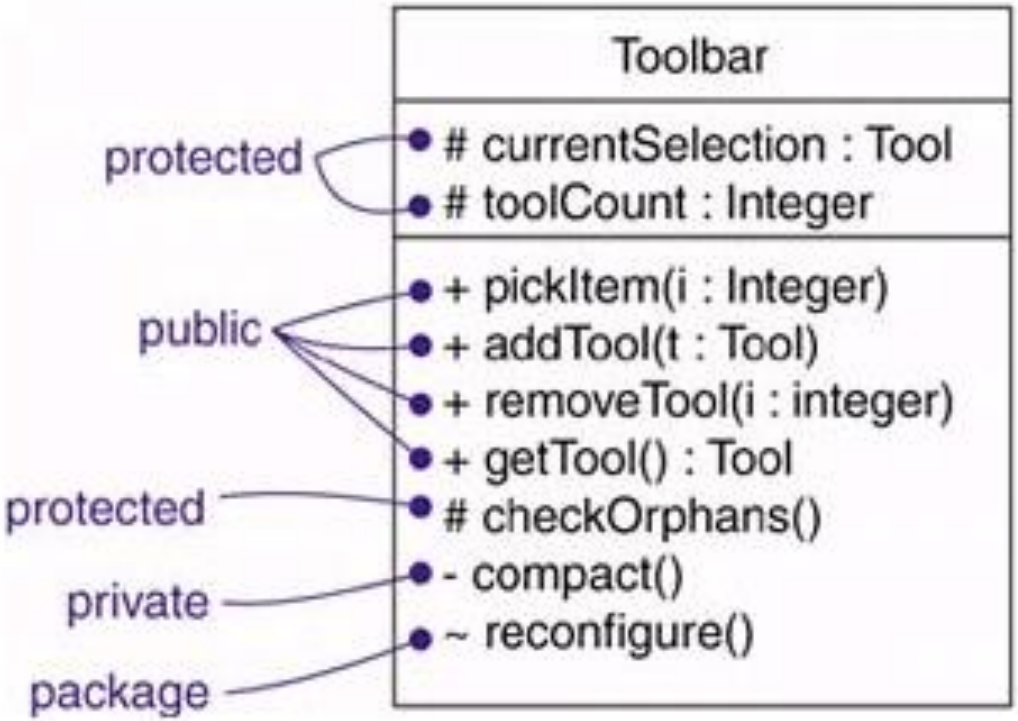
Figure . Operations and Their Signatures

4.6.2 类(Class)

- 访问属性
  - public：　任何类都可访问该类。 +.
  - protected：该类的任何后代都可访问该类。#.
  - private：只有该类可访问。 - .
  - Package：只有在同一个包中的类可访问。 ~.

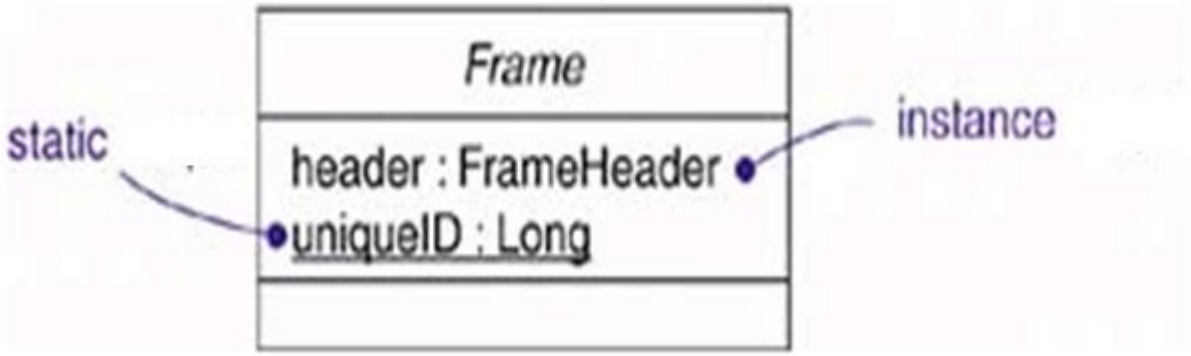
4.6.2 类(Class)

- 访问属性



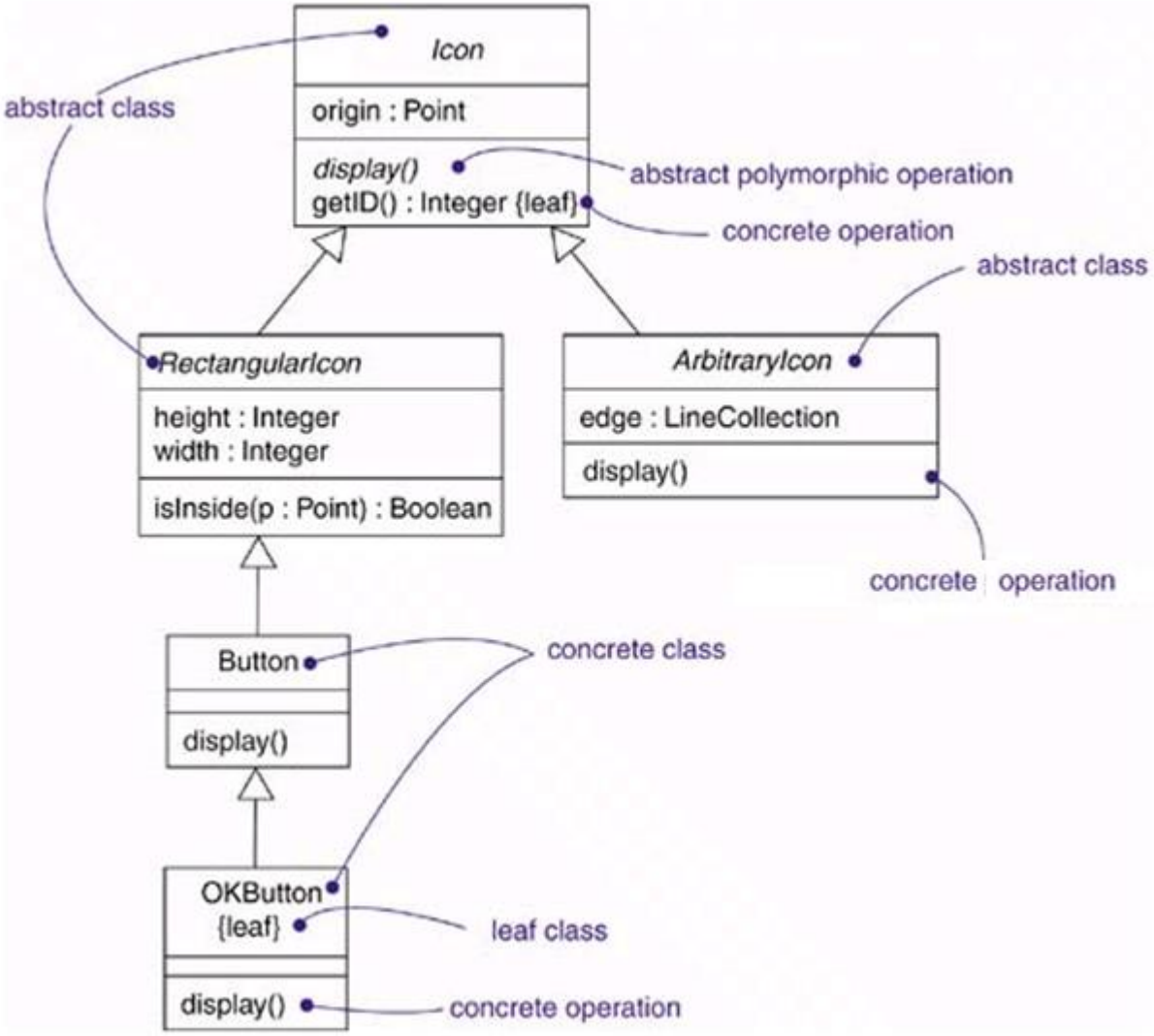
4.6.2 类(Class)

- 实例变量和静态变量
  - 实例变量（instance）：
    类的每个实例都拥有该属性的 值。
  - 静态变量（static ）
    类的每个实例都共享该属性的值。



4.6.2 类(Class)

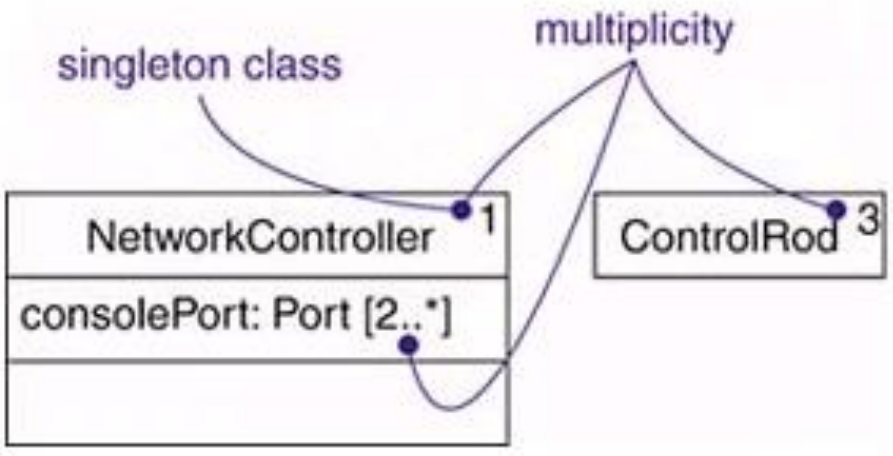
- 抽象 (Abstract), 叶子(Leaf), 多态(Polymorphic)元素



4.6.2 类(Class)

- 类的多样性(Multiplicity)

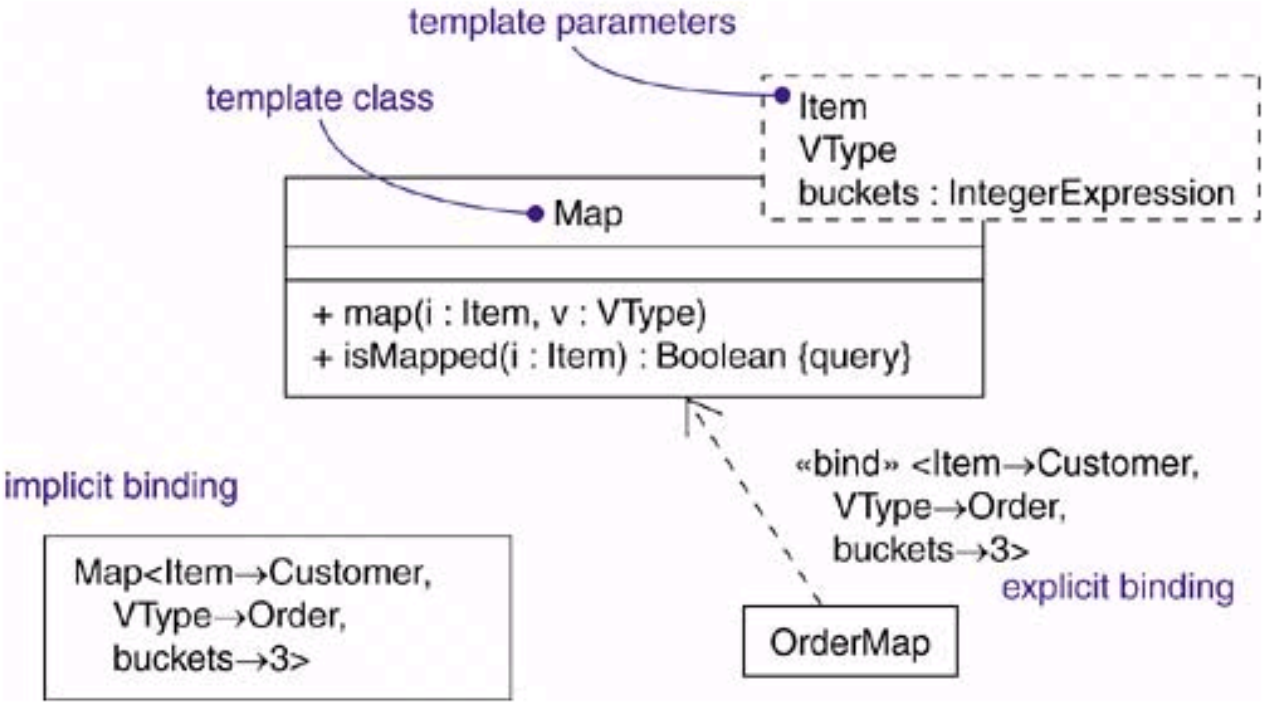
  类实例的数目。



多样性

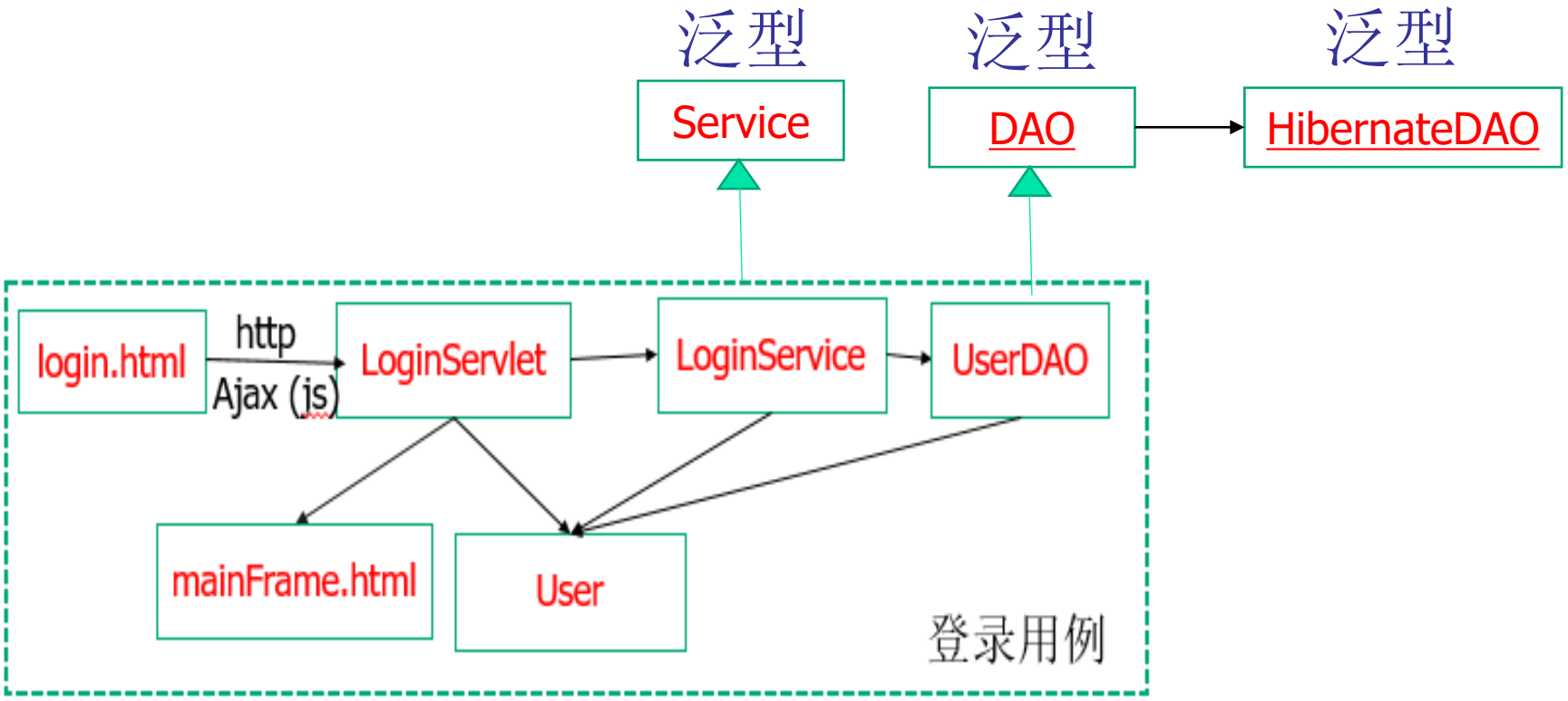
4.6.2 类(Class)

- 类模板(Template Classes)
  - ❖不能直接使用类模板，而必须要先实例化它。
  - ❖实例化过程：
    用实际参数代替模板参数(template parameters)



4.6.2 类(Class)

类模板例子：



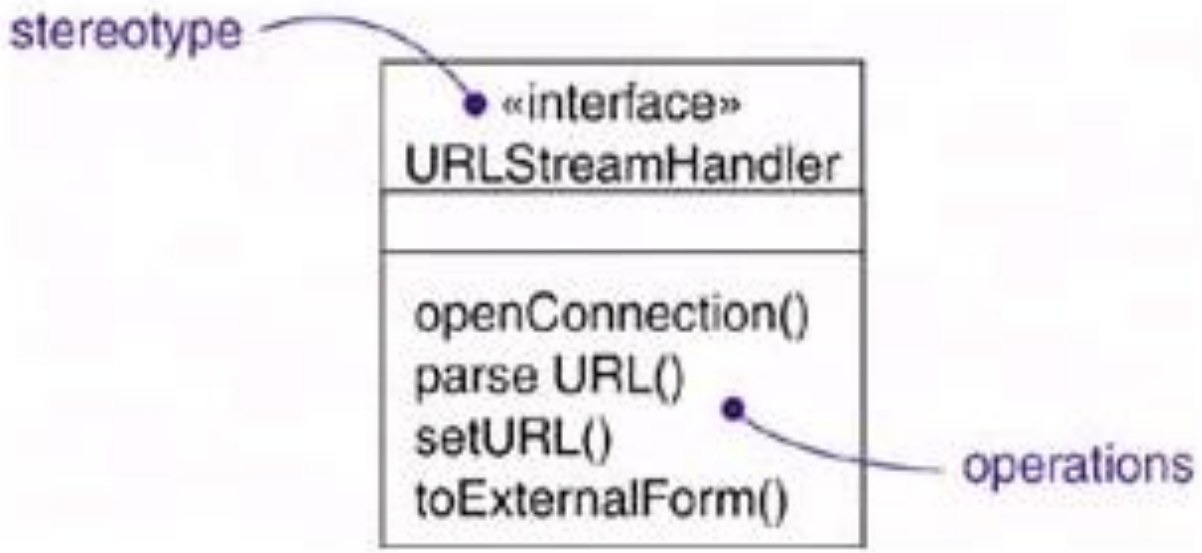
也可以抽象出一个Service父类（泛型)

4.6类图

4.6.3 接口(Interface)

- 定义

接口是操作的集合，这些操作规范了类或者组件的服务

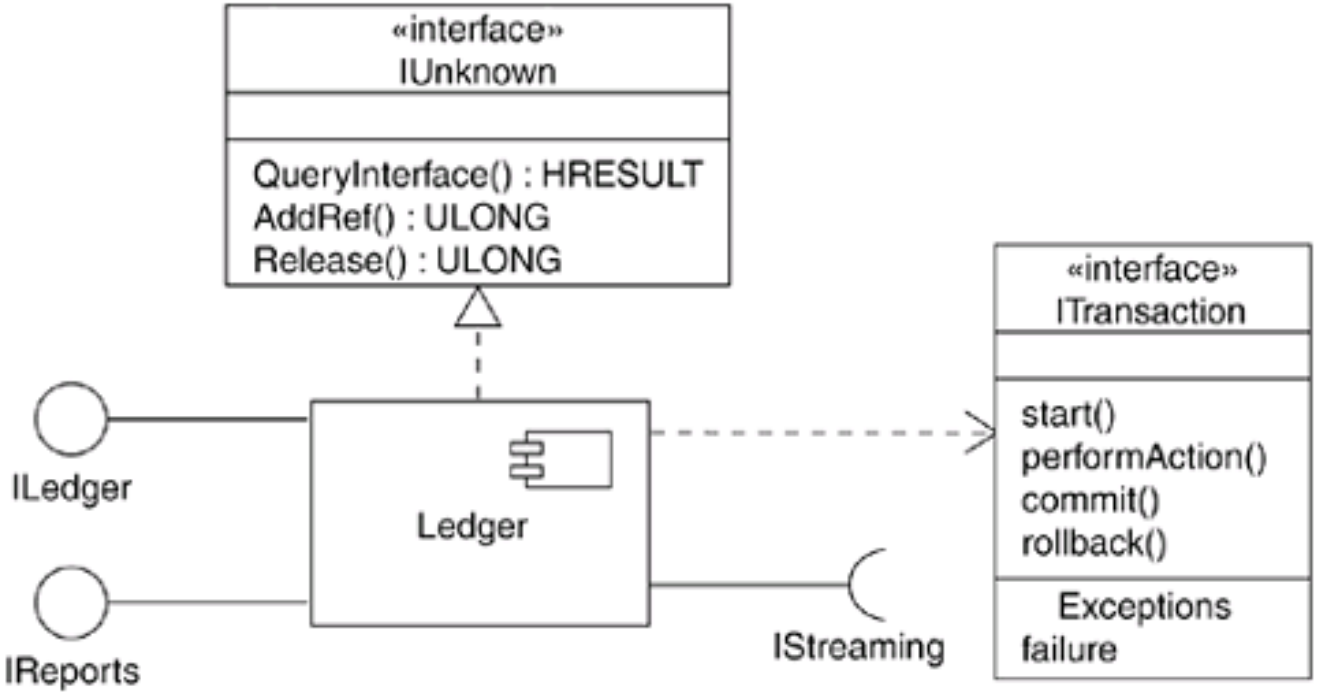


4.6类图

4.6.3 接口(Interface)

- 建模接口的原因

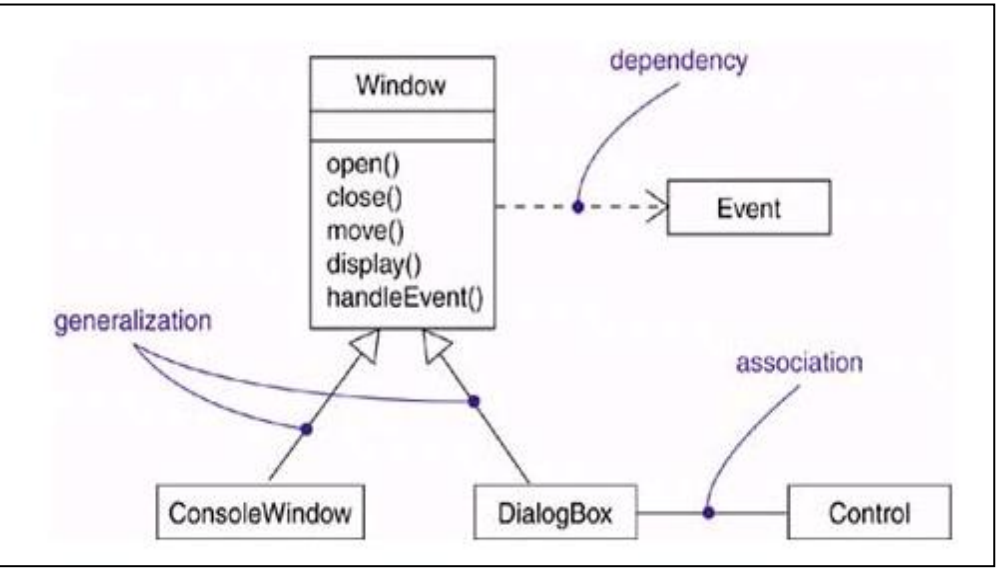
在基于组件的软件系统中，提供这些组件间的衔接(seams)。例如： Eclipse, .NET, or Java Beans.

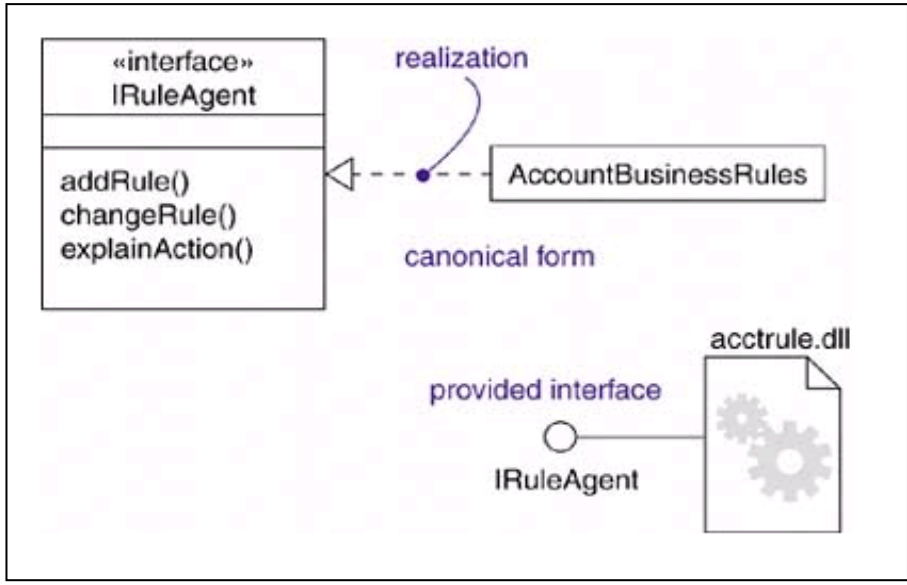


4.6类图

4.6.4关系(relationship)

- 种类：
  - 依赖(dependency)
  - 泛化(generalization)
  - 关联(association)
  - 实现(realization)



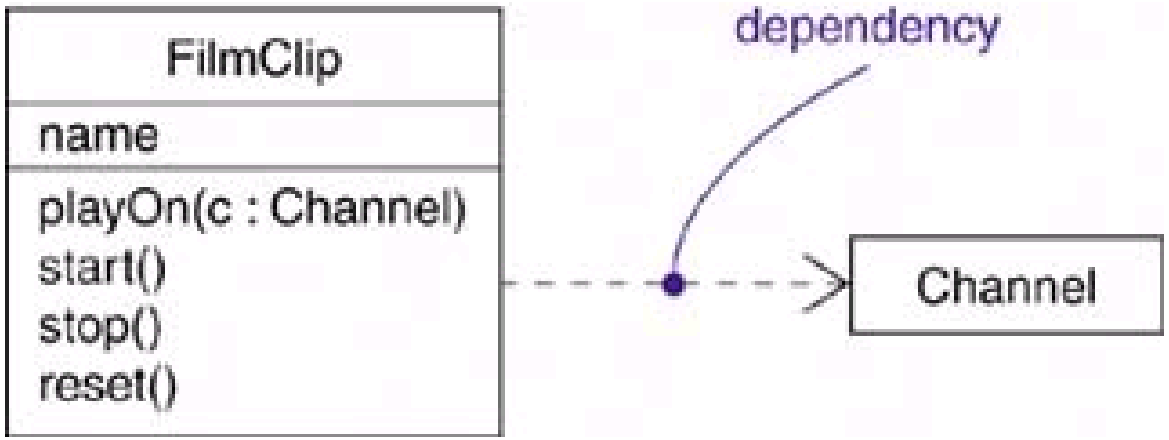


4.6类图

4.6.4关系(relationship)

- 依赖(dependency)

A *dependency* is a relationship that states that **one thing uses** the information and services of **another thing**, but not necessarily the reverse.



```
package util.jdbc;
import java.sql.*;
public class  DBJDBCConnection{
    public DBJDBCConnection(){
            super();
      }
    public Connection getConnection(String driver, String url, String user,String
pwd) {
                    Connection conn;
                    try{
                            Class.forName(driver);
                            conn=DriverManager.getConnection(url, user, pwd);
                            return conn;
                          }catch(ClassNotFoundException e){
                                    System.out.println(e.getMessage());
                            }
                           catch( SQLException e){
                            System.out.println(e.getMessage());
                        }
                    return null;
    }
}
```

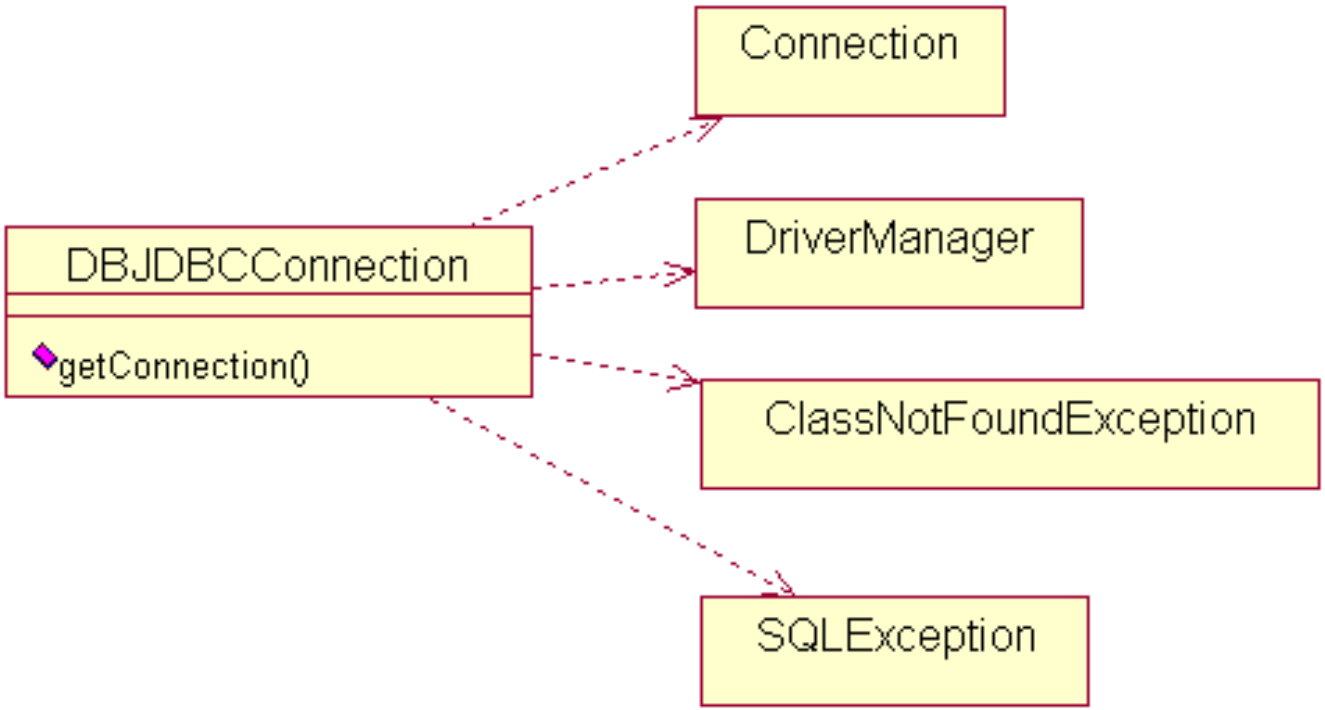

Connection
DriverManager
DBJDBCConnection
getConnection()
ClassNotFoundException
SQLException

图 依赖关系建模

4.6.4关系(relationship)

- 依赖(dependency)
  - 依赖关系种类:

Table 4-3: *Kinds of Dependencies*

Dependency	*Function*	*Keyword*
access	Permission for a package to access the contents of another package	access
binding	Assignment of values to the parameters of a template to generate a new model element	bind
call	Statement that a method of one class calls an operation of another class	call
derivation	Statement that one instance can be computed from another instance	derive
friend	Permission for an element to access the contents of another element regardless of visibility	friend
import	Permission for a package to access the contents of another package and add aliases of their names to the importer's namespace	import
instantiation	Statement that a method of one class creates instances of another class	instantiate

4.6.4关系(relationship)

- 依赖(dependency)
  - 依赖关系种类:

parameter	Relationship between an operation and its parameters	parameter
realization	Mapping between a specification and an implementation of it	realize
refinement	Statement that a mapping exists between elements at two different semantic levels	refine
send	Relationship between the sender of a signal and the receiver of the signal	send
trace	Statement that some connection exists between elements in different models, but less precise than a mapping	trace
usage	Statement that one element requires the presence of another element for its correct functioning (includes call, instantiation, parameter, send, but open to other kinds)	use

4.6.4关系(relationship)

- 依赖(dependency)
  - 类图中类(class)、对象(object)间的依赖关系

1. bind
2. derive
3. permit

类与类

4. instanceOf
5. instantiate

类与对象

6. powertype
7. refine
8. use

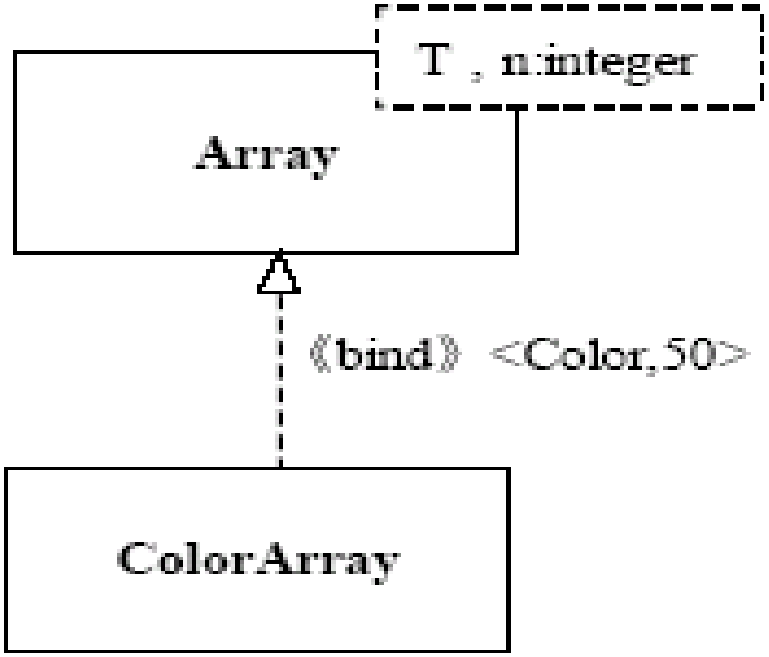
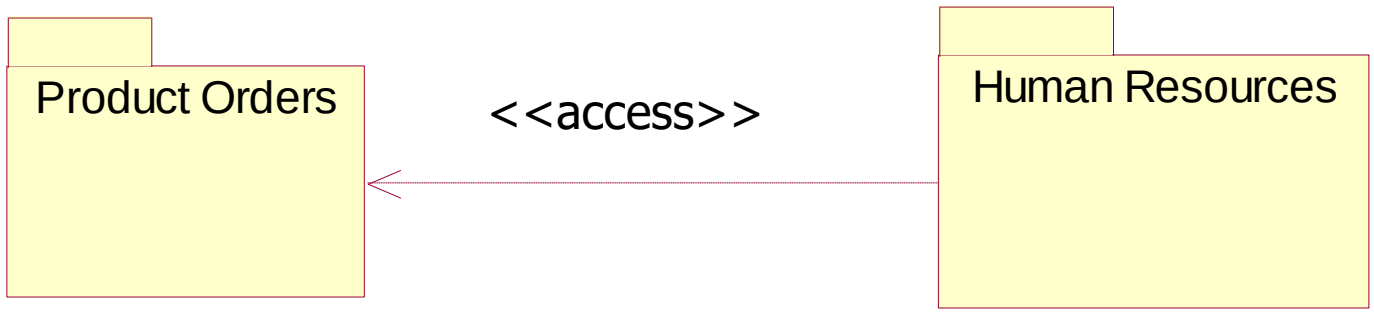


Figure- bind

Figure. refine

4.6.4关系(relationship)

- 依赖(dependency)
  - 包(packages)间的依赖关系。
    1. import
    2. access



4.6.4关系(relationship)

- 依赖(dependency)
  - 用例(use cases)间的依赖关系：
    1. extend
    2. include
  - 对象交互(interactions)过程中的依赖关系：
    1.send
  - 子系统(subsystems)中各元素间的依赖关系:
    1.Trace

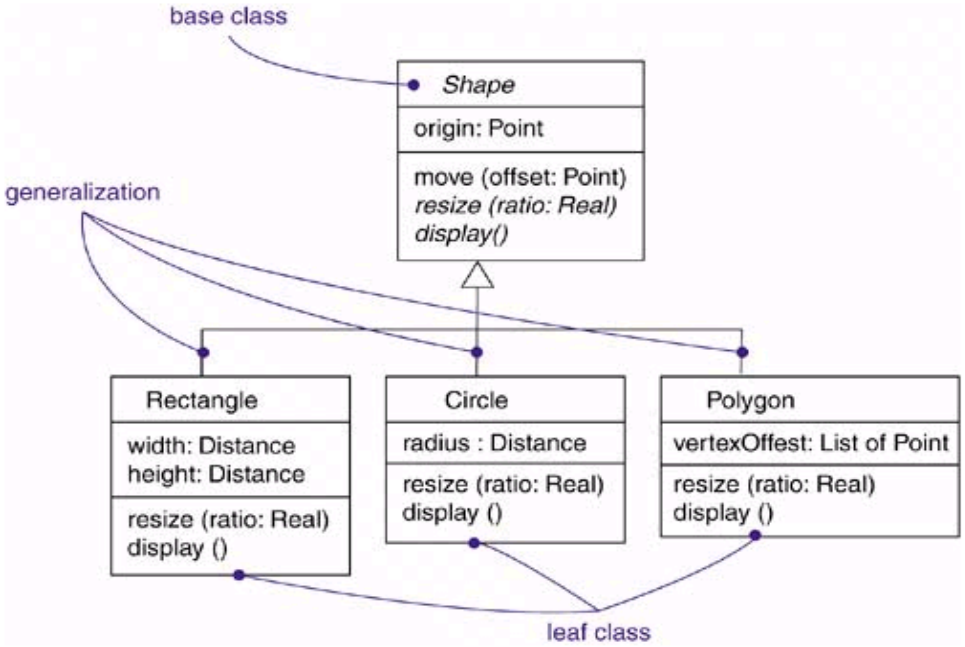
4.6.4关系(relationship)

- 说明

　　所有的关系(泛化、关联、实现)从广义上讲都是一种依赖关系。

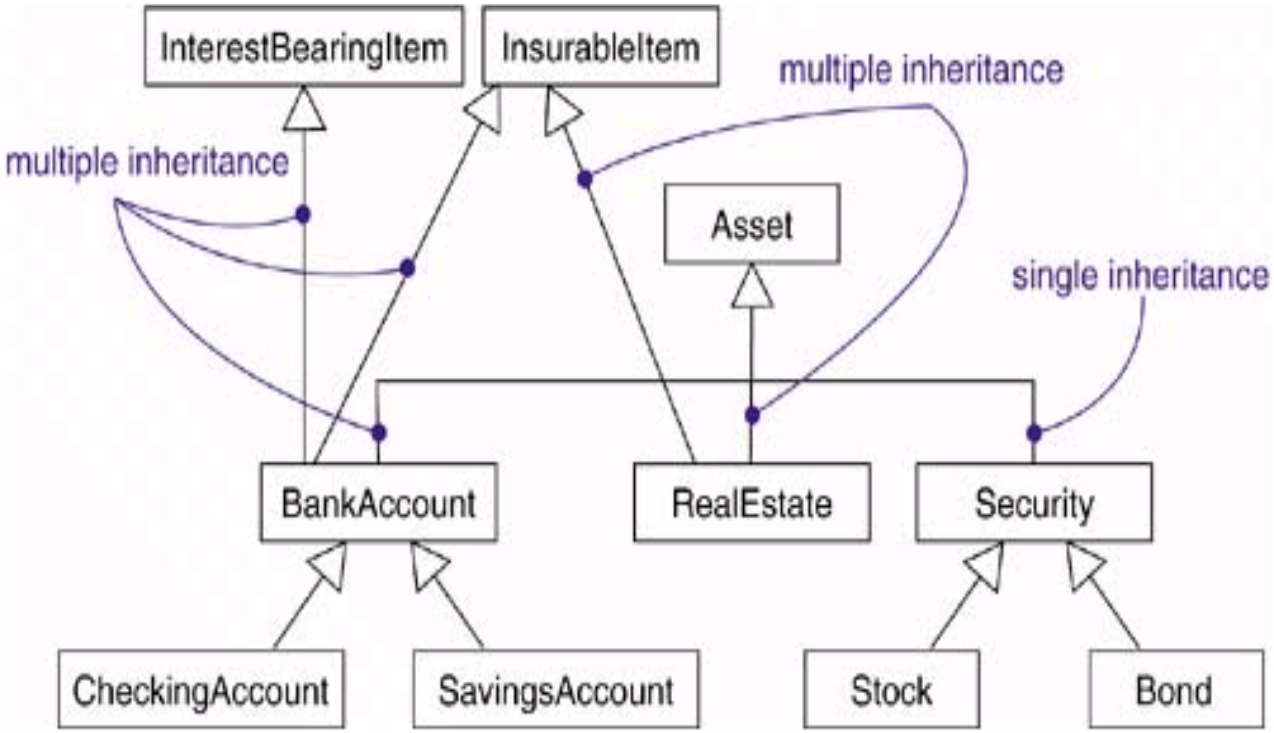
4.6.4关系(relationship)

- 泛化(Generalizations)
  - 一般与特殊的关系。 general / specific
  - inheritance关系
  - parent/child关系
  - "is-a-kind-of"关系



4.6.4关系(relationship)

- 泛化(Generalizations)
  - 继承种类
    - 单继承(single inheritance): 一个类只有一个父类
    - 多继承(multiple inheritance):一个类可不只有一个父类



4.6.4关系(relationship)

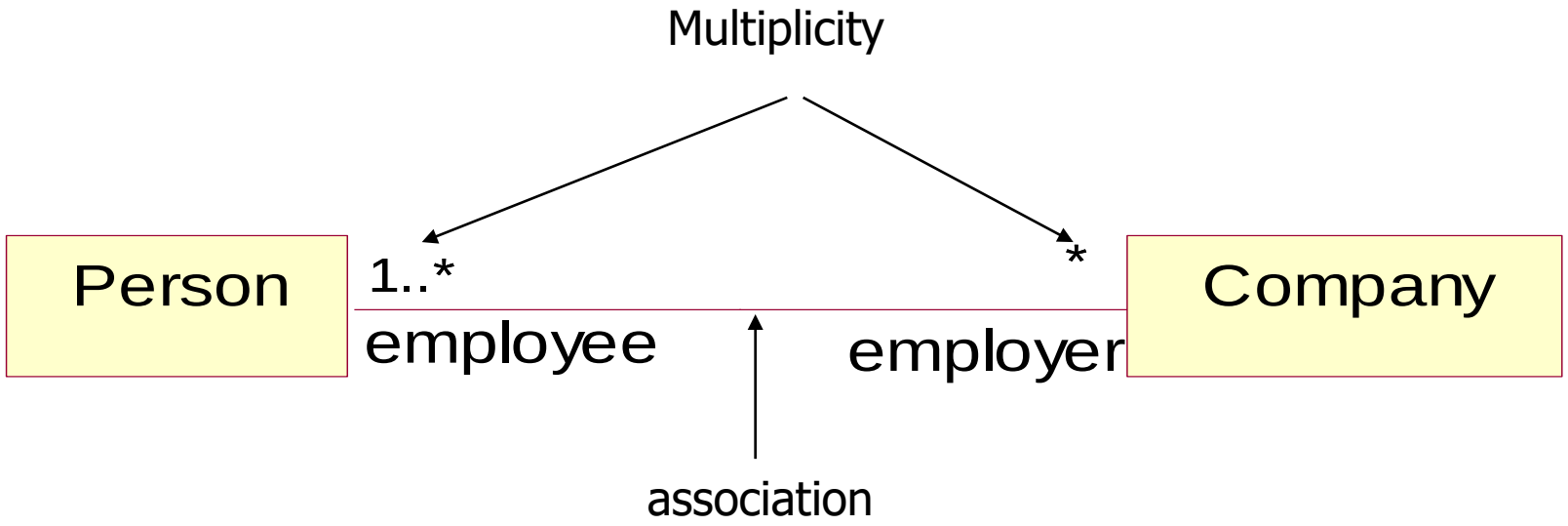
- 关联(Associations)

An *association* is a **structural relationship** that specifies that objects of one thing are connected to objects of another.

4.6.4关系(relationship)

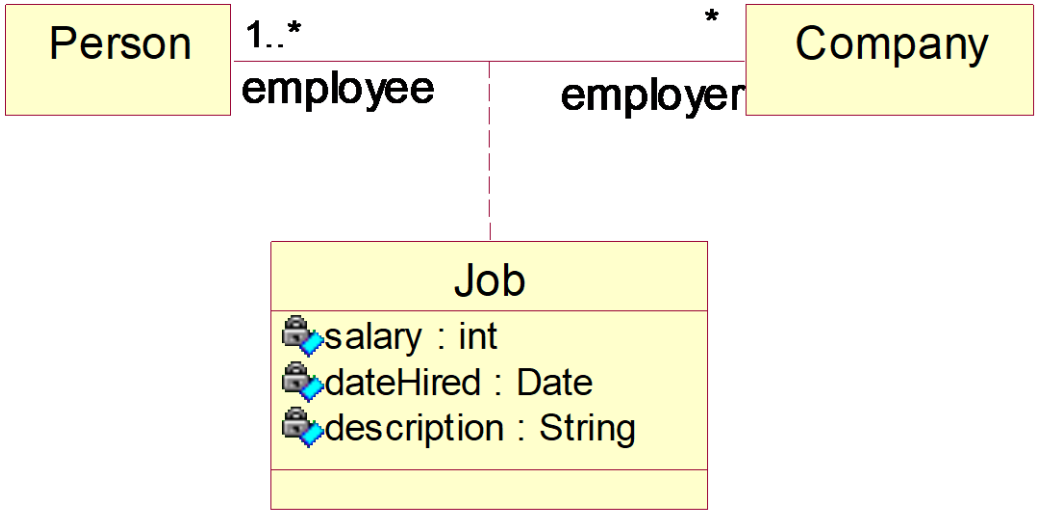
- 关联(Associations)
  - 多样性(Multiplicity)

    在关联关系实例中，有多个对象相连。



4.6.4关系(relationship)

- 关联(Associations)
  - 关联类(Association Classes)
    - 在两个类发生关联关系中，关联本身也会有自身的属性。
    - 在关联类中，同时都具备有关联关系本身的属性和关联类的属性。




```
public class Job
{
    private int salary;
    private Date dateHired;
    private String description;
    Company employer;
    Person  employee;
}
```

4.6.4关系(relationship)

- 实现(Realizations)
  - interface与class之间，或者interface与component间。Class与component提供了interface的实现。

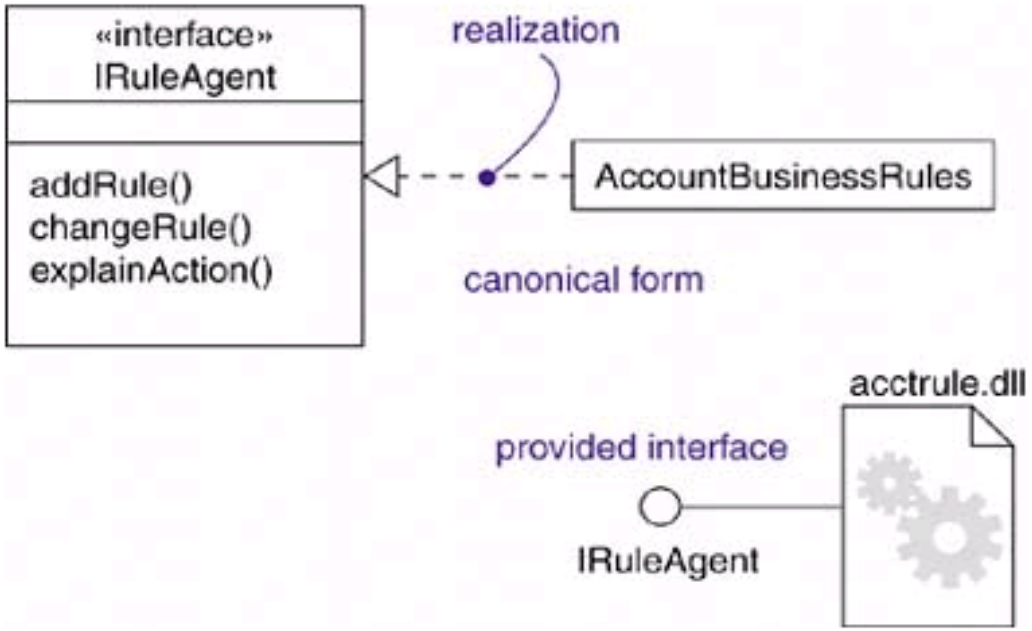
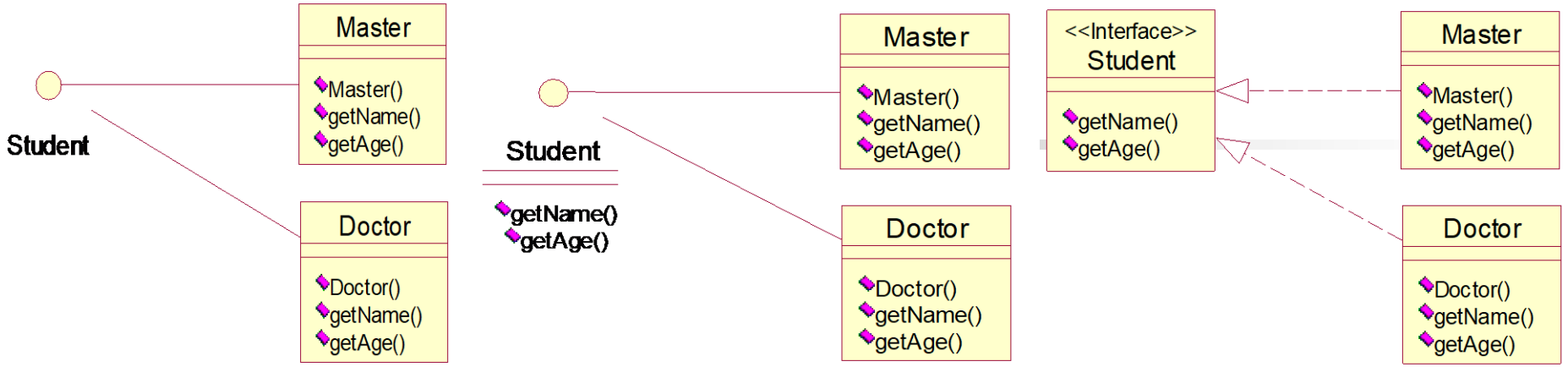


Figure . Realization of an Interface




```
public interface Student {
        public String getName();
        public int getAge();
}
```

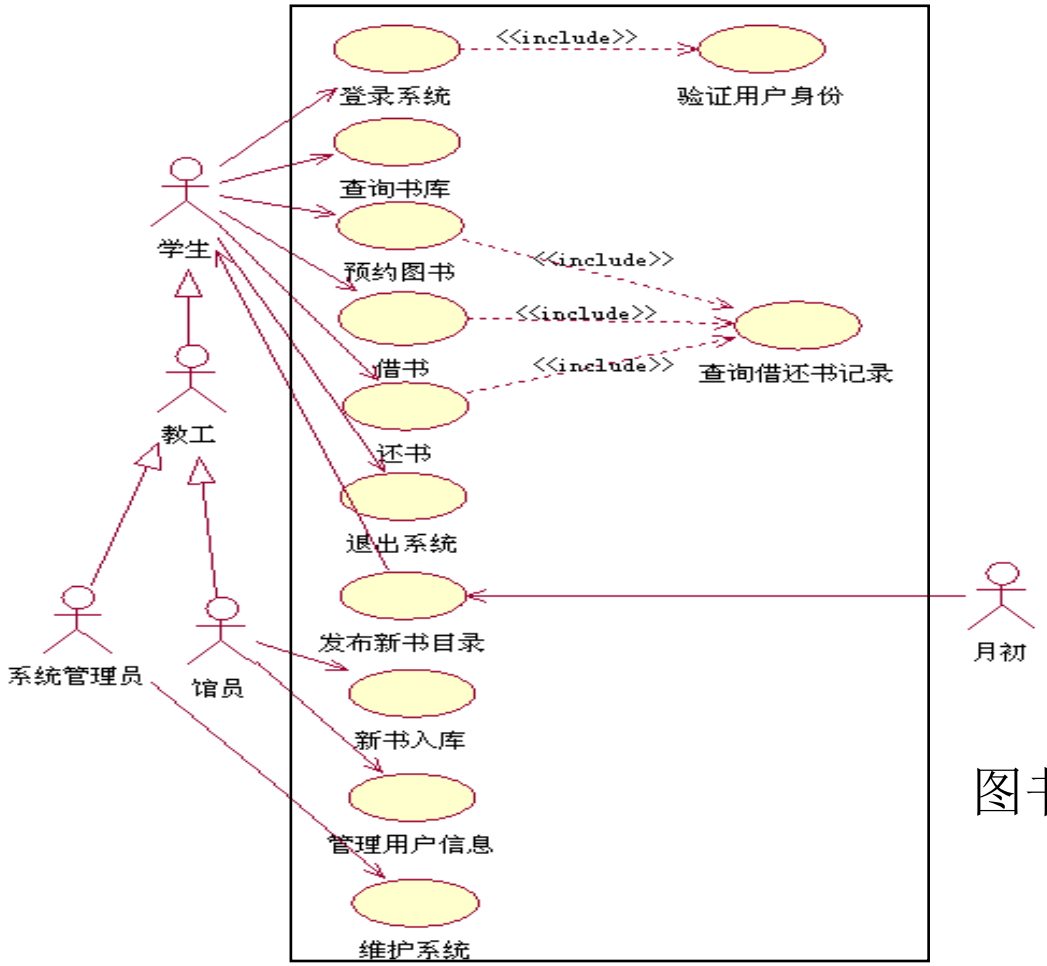
```
public class Master implements Student {
  public Master() {   }
  public String getName() { return null;  }
  public int getAge() { return 0;  }
}
```

```
public class Doctor implements Student {
  public Doctor() {    }
   public String getName() {   return null; }
   public int getAge()    {    return 0;   }
}
```

4.6类图

4.6.5 领域建模（类图）

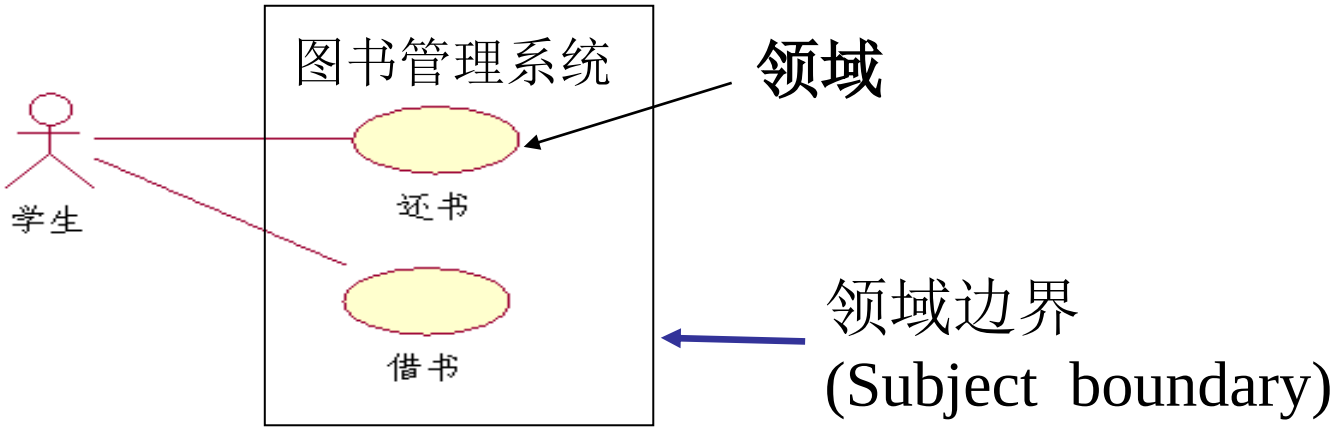
- Case-图书管理系统领域建模



图书管理系统

4.6.5 领域建模（类图）

- 领域(subject): 系统或者子系统。由用例集描述。



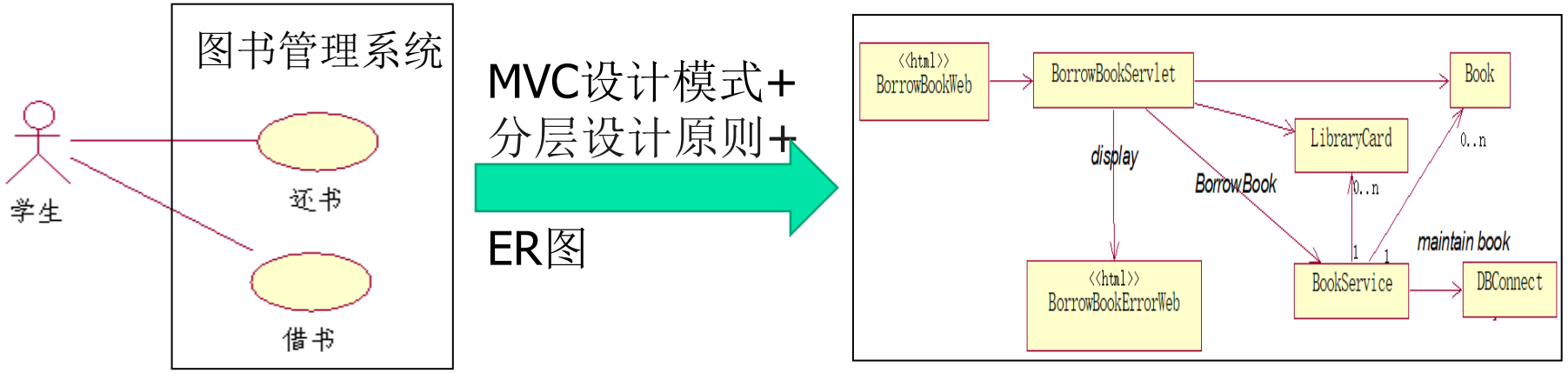
- 用例代表领域的功能（行为）。
- 活动者代表与领域交互的一方。
- 所有的用例描述了领域全部的功能（行为）。

4.6.5 领域建模（类图）

如何抽象出(设计）领域的类？

即：如何设计系统（子系统/用例）的类？

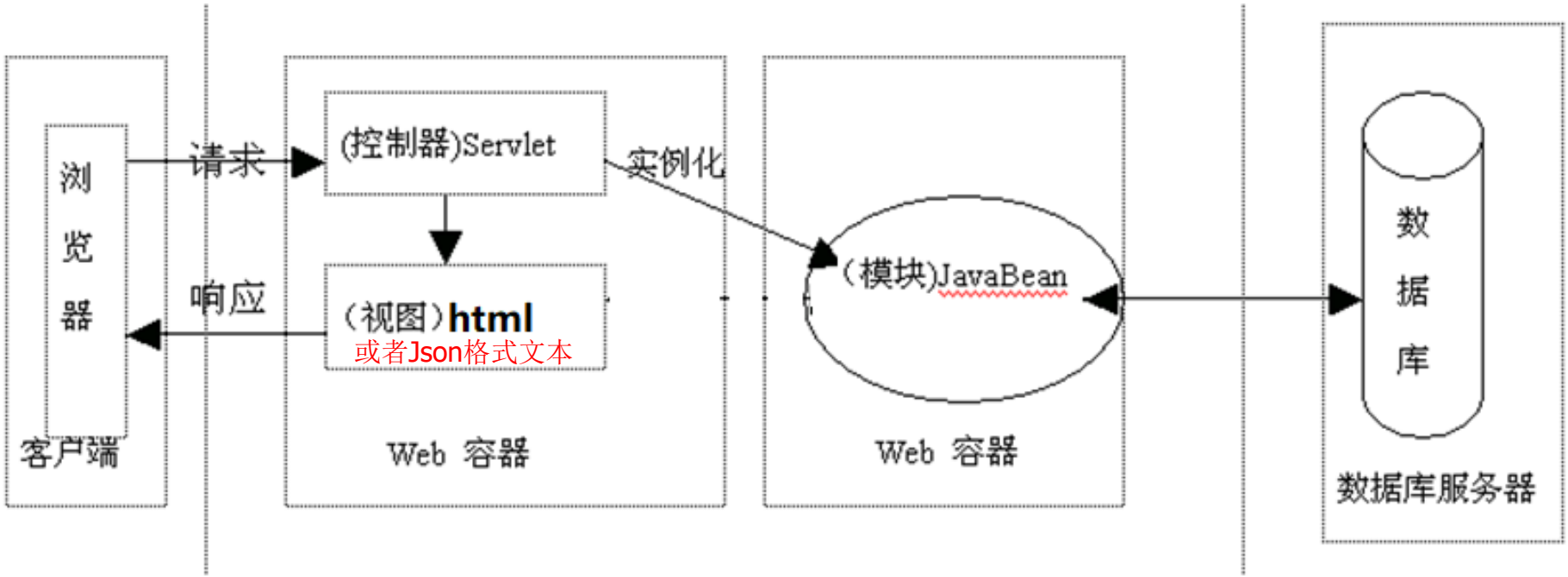
- 基于用例驱动的面向对象软件分析与设计
  - 管理信息系统(MIS)
  - 具体的软件体系结构风格
  - 针对用例图中每一个用例建模**领域模型**（类）



4.6.4 领域建模（类图）

- 修改后的 Jsp Model 2 模型

MVC设计模式和Web应用系统



4.6.4 领域建模（类图）

- 领域建模（类图）

  对于每一个用例（子系统）都有相应的：

  - 控制类(C)： 只起到**页面跳转**作用
  - 视图类(V)： **界面**
  - 模型类(M) ： **ER图**
  - 服务类(Service)：**业务**（用例文本描述的事件流/活动图）、数据库增、删、改、查询。
  - 访问数据库类(DAO)：**数据库增、删、改、查询**

4.6.4 领域建模（类图）

- 例：用户登录用例

需求分析(用户提供)：

1. 用户登录用例界面：

2. 用户登录用例描述：

(1) 用户在登录界面输入用户名、密码.

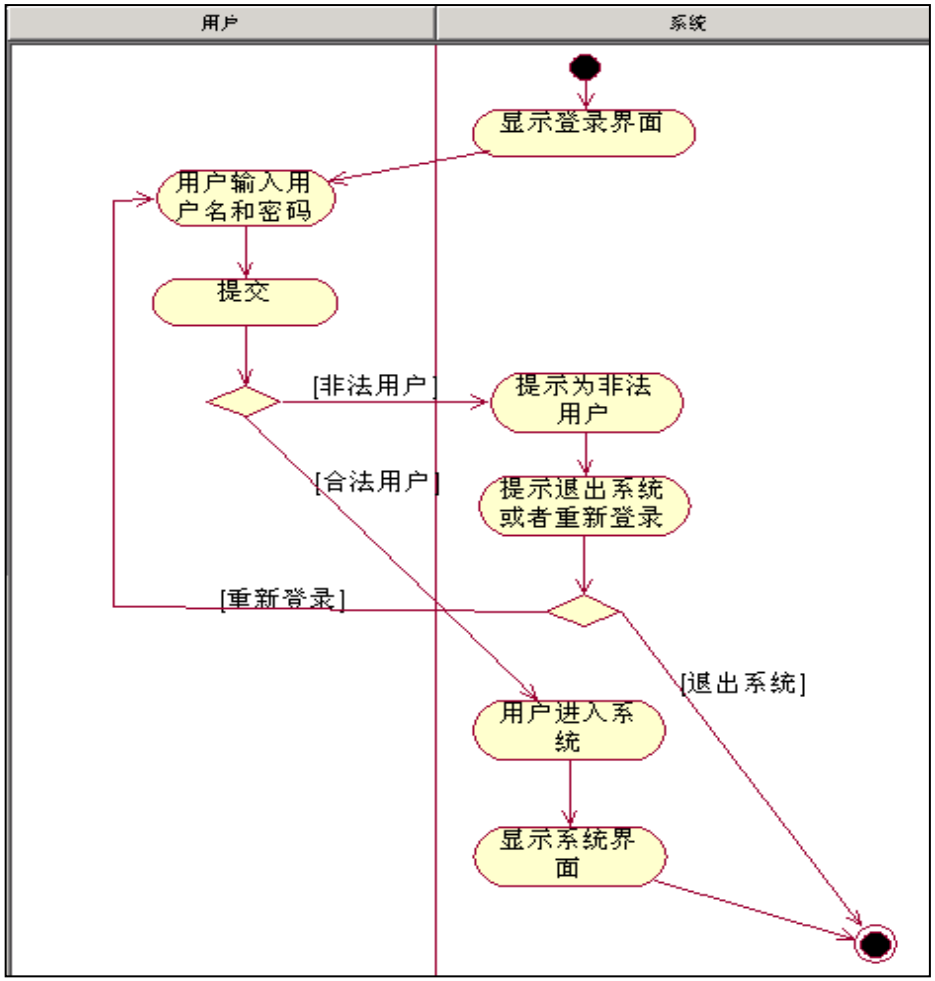
(2) 如果登录失败，则显示：您填写的用户名或密码错误。

(3)如果登录成功，则进入系统界面 .

4.6.4 领域建模（类图）

- 例：用户登录用例

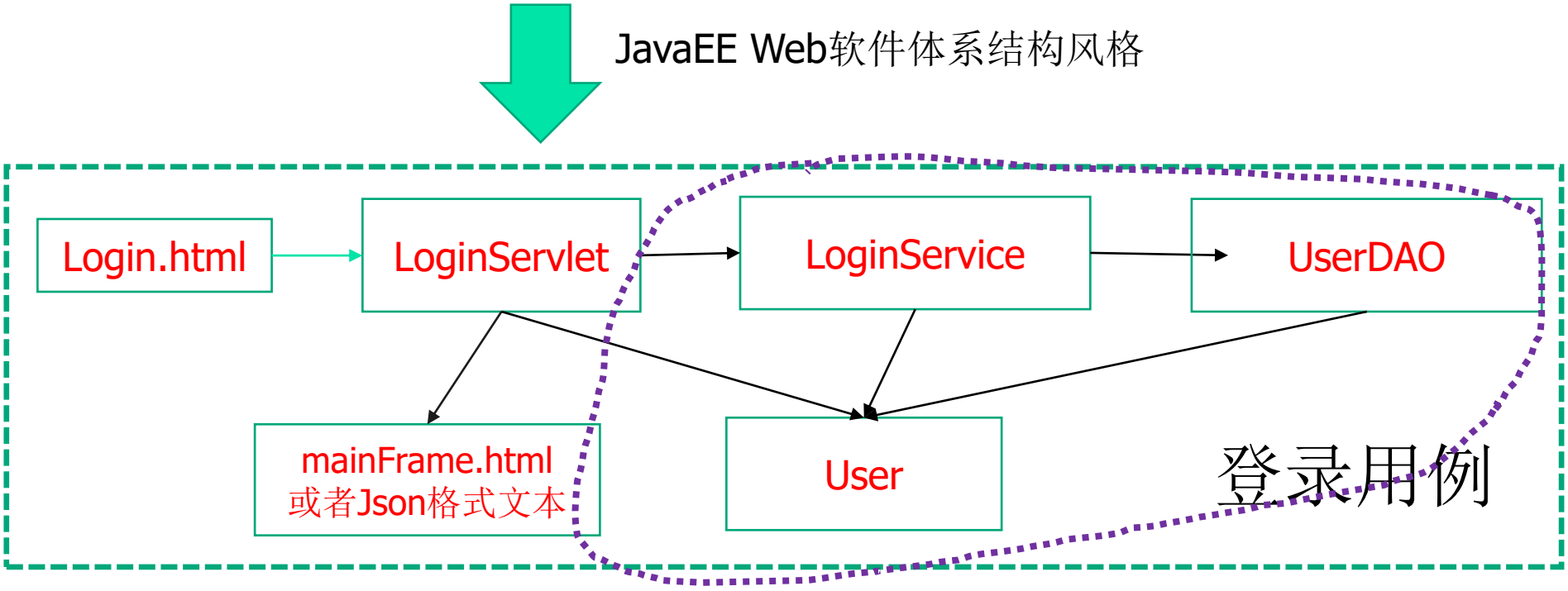
3. 用户登录用例活动图：



4.6.4 领域建模（类图）

- 例：用户登录用例

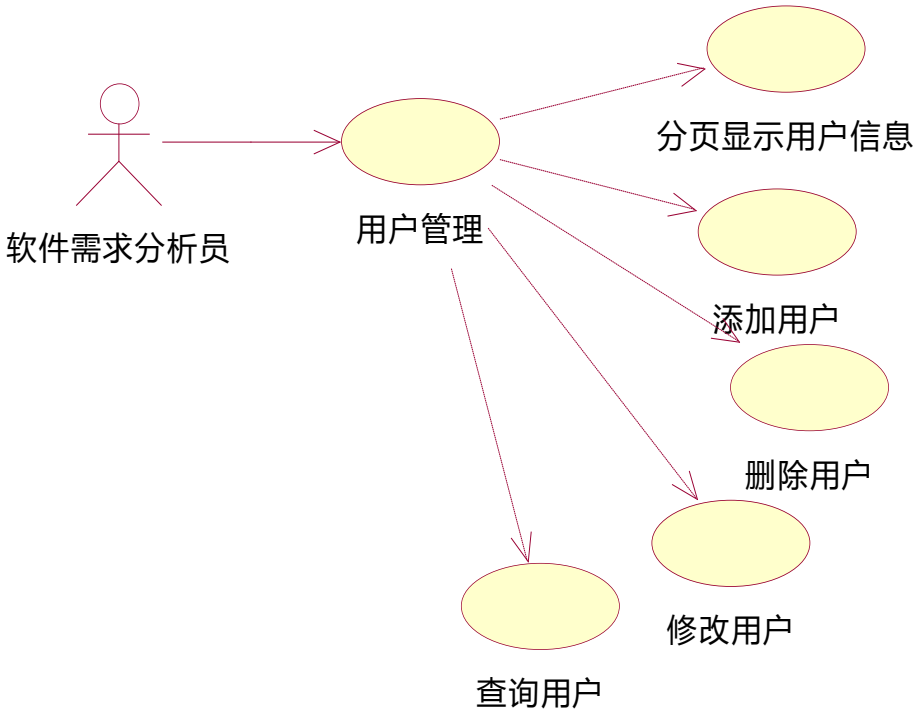
4.网页登录用例(login)领域建模



完善登录用例涉及的类及类间关系

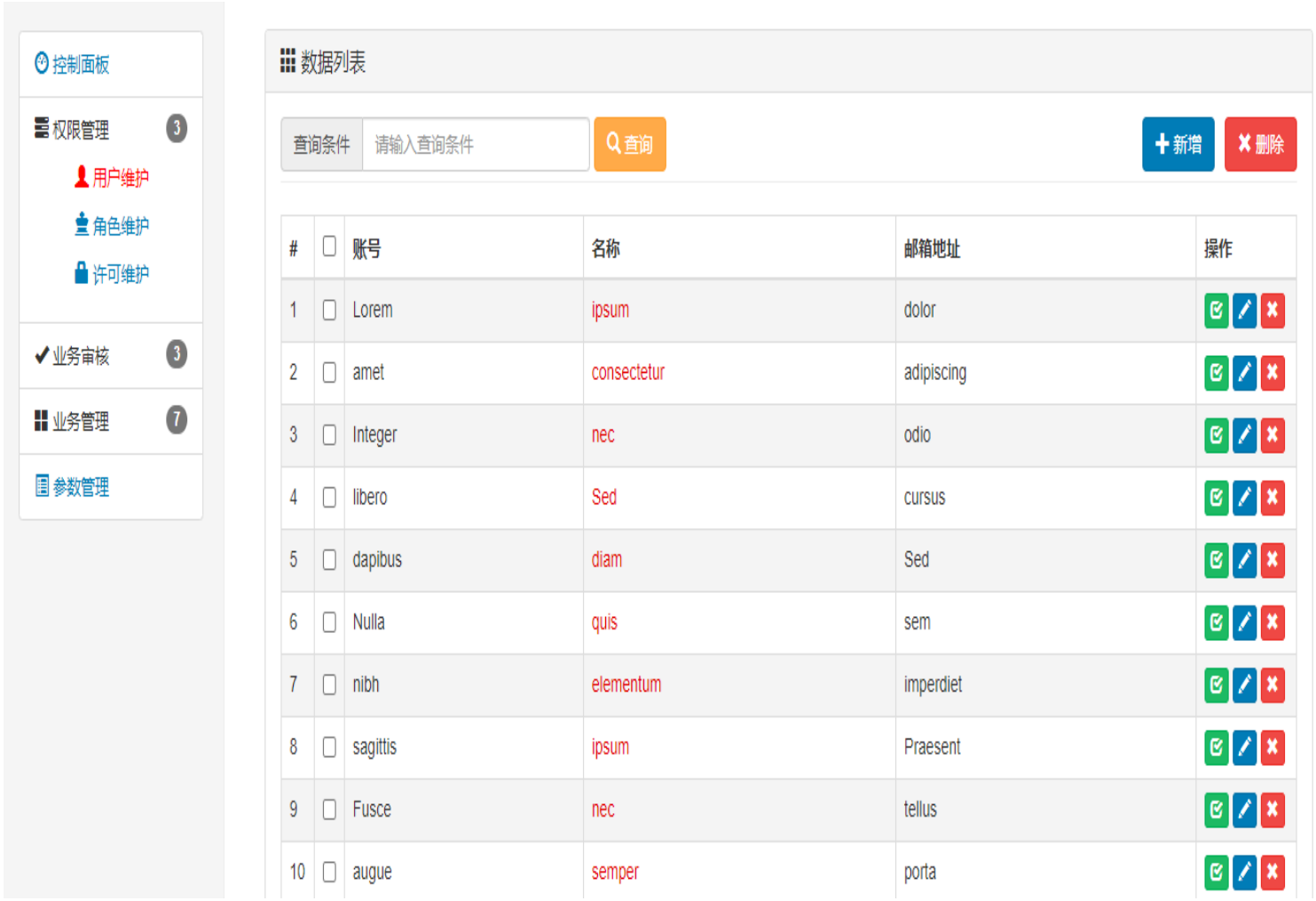
4.6.4 领域建模（类图）

- 例：添加用户用例 ：用户管理用例的一个子用例



4.6.4 领域建模（类图）

- 用户维护主界面（用户管理）



4.6.4 领域建模（类图）

- 添加用户界面

- 添加用户用例（文本描述）

前置条件：一个合法的需求分析人员已经登录系统

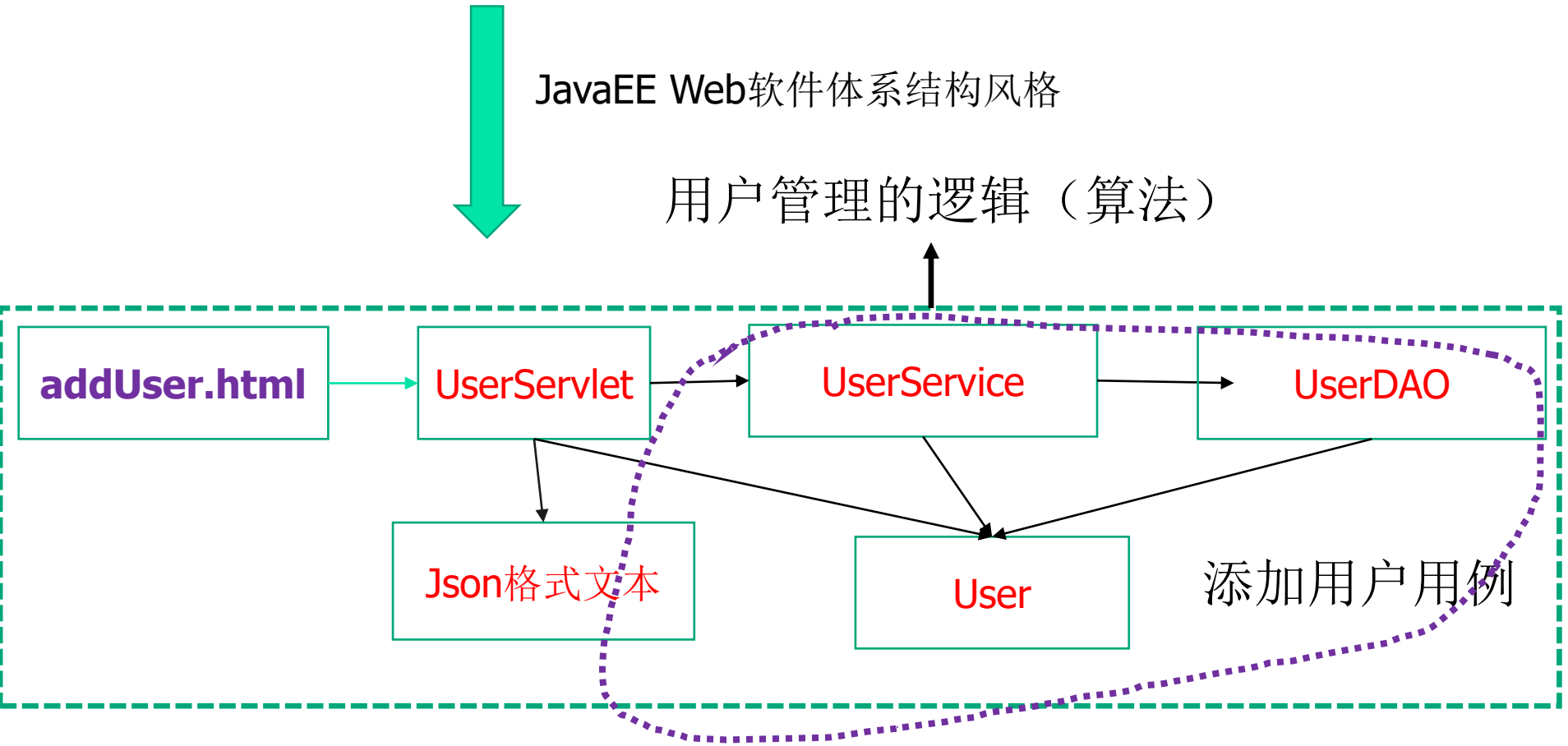
事件流：

1. 当需求分析人员在用户维护主界面选择新增按钮时,用例开始。
2. 系统显示新增用户界面。
3. 需求分析人员输入该用户的用户名、邮箱
4. 点击保存按钮。

后置条件：系统保存新增的用户（密码默认888888）。

4.6.4 领域建模（类图）

- 例：添加用户用例

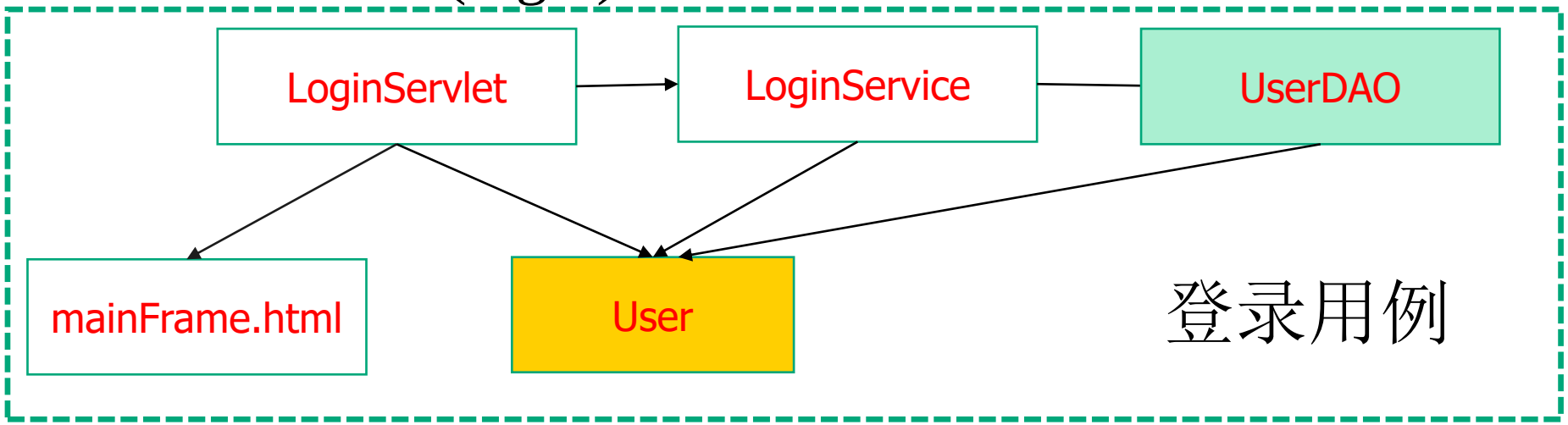


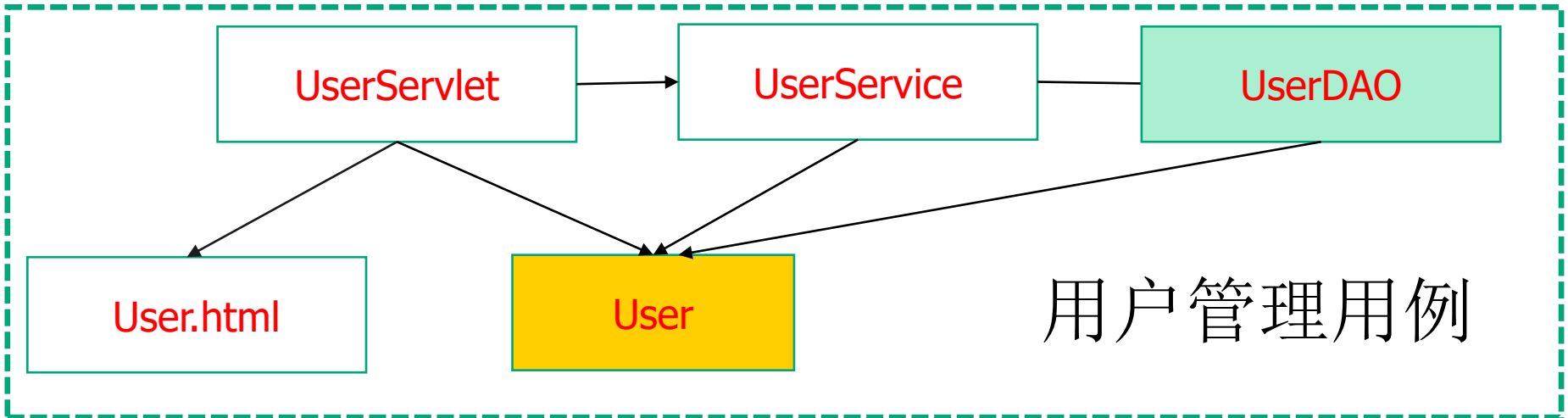
添加用户用例涉及的类及类间关系

4.6.4 领域建模（类图）

- 完善用例间的类及类间关系

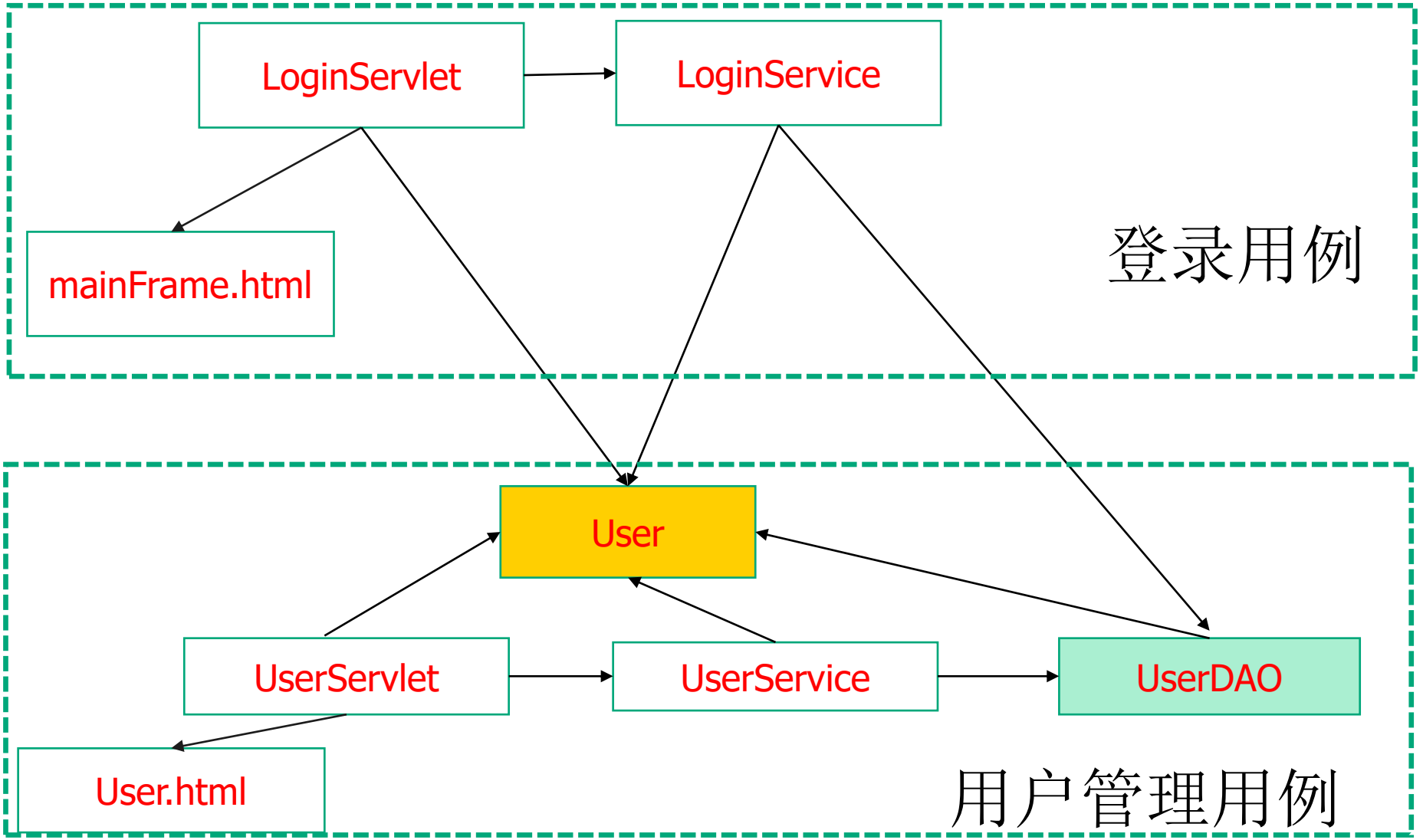
网页登录用例(login)和用户管理用例间类的关系





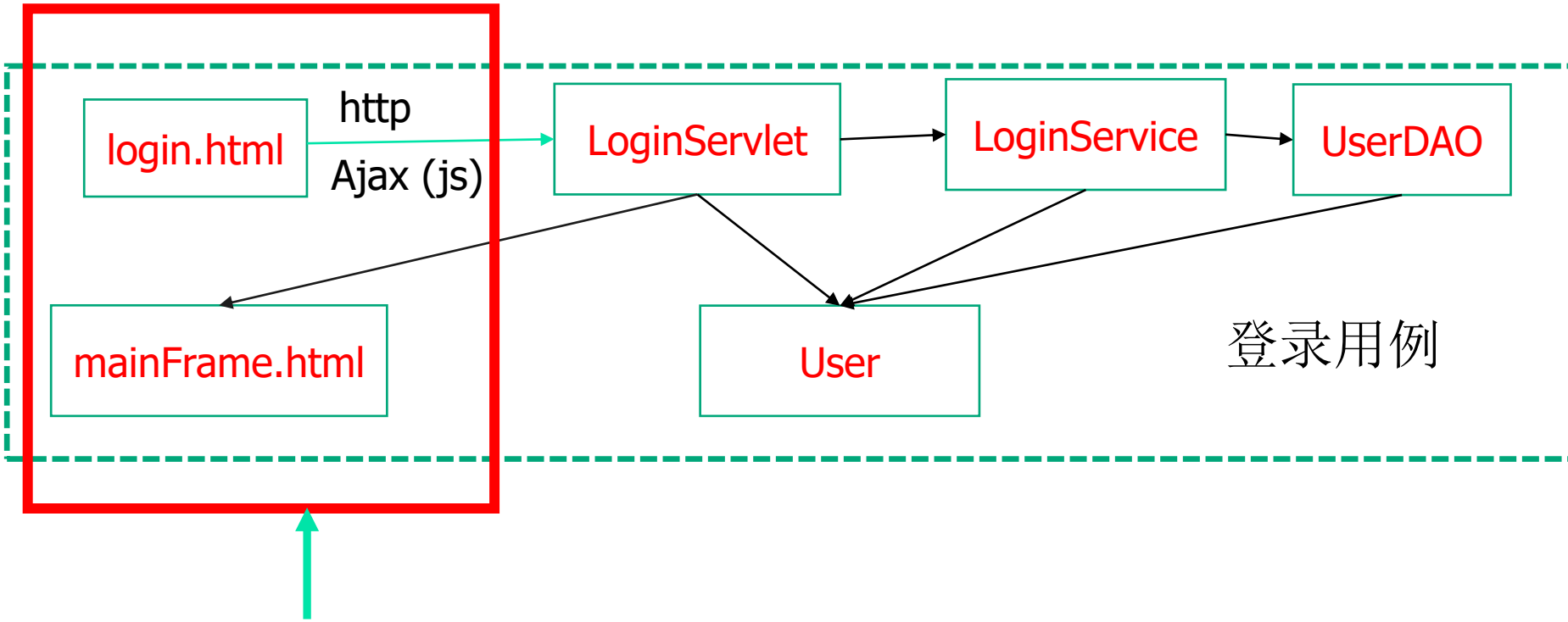
4.6.4 领域建模（类图）

- 完善用例间的类及类间关系



4.6.4 领域建模（类图）

- 例： 用户登录用例 (Android实现: Web方式)

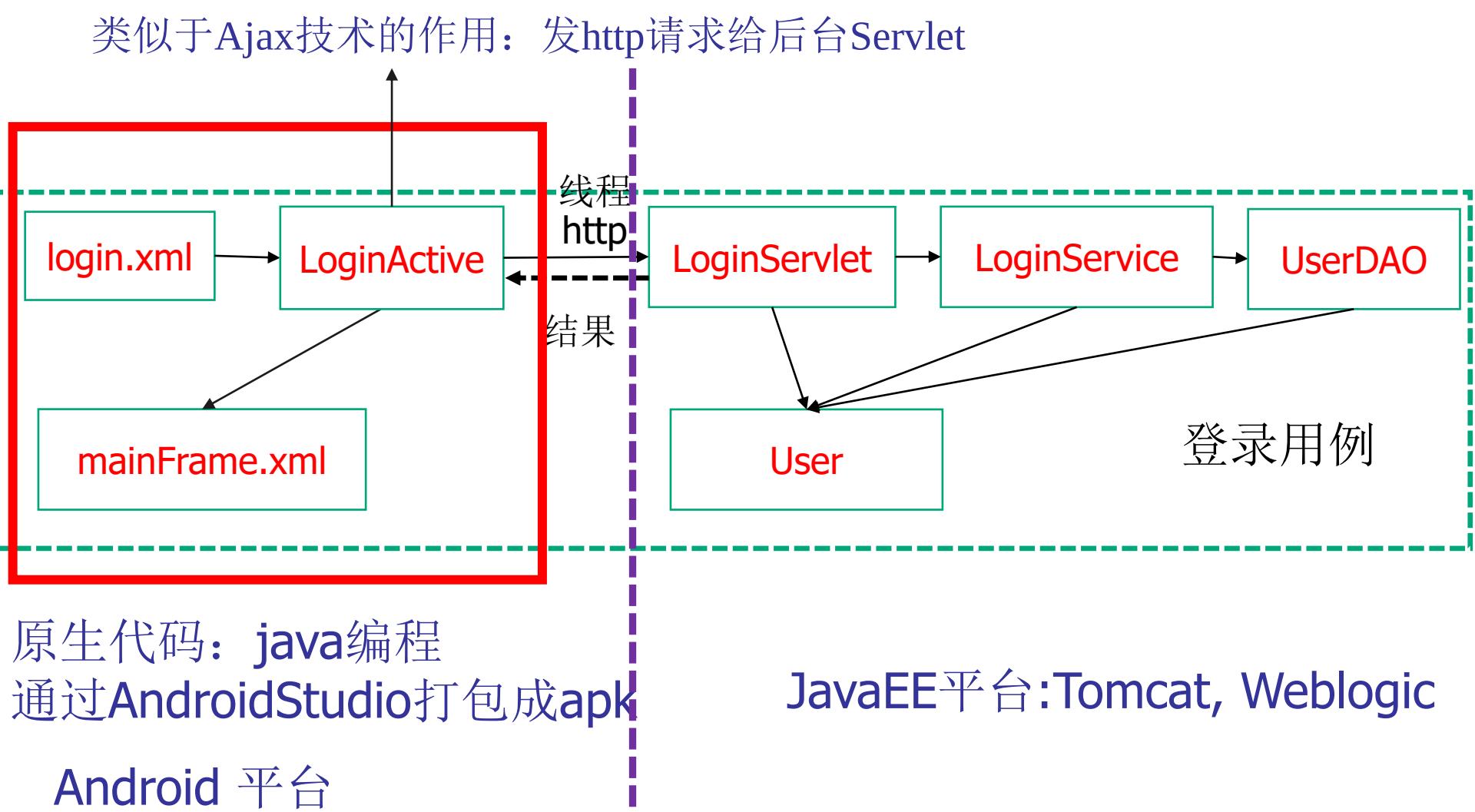


用打包软件(Hbuilder)打包所有的html文件、javascript文件打包成apk,发布到Android手机上。

login.apk

4.6.4 领域建模（类图）

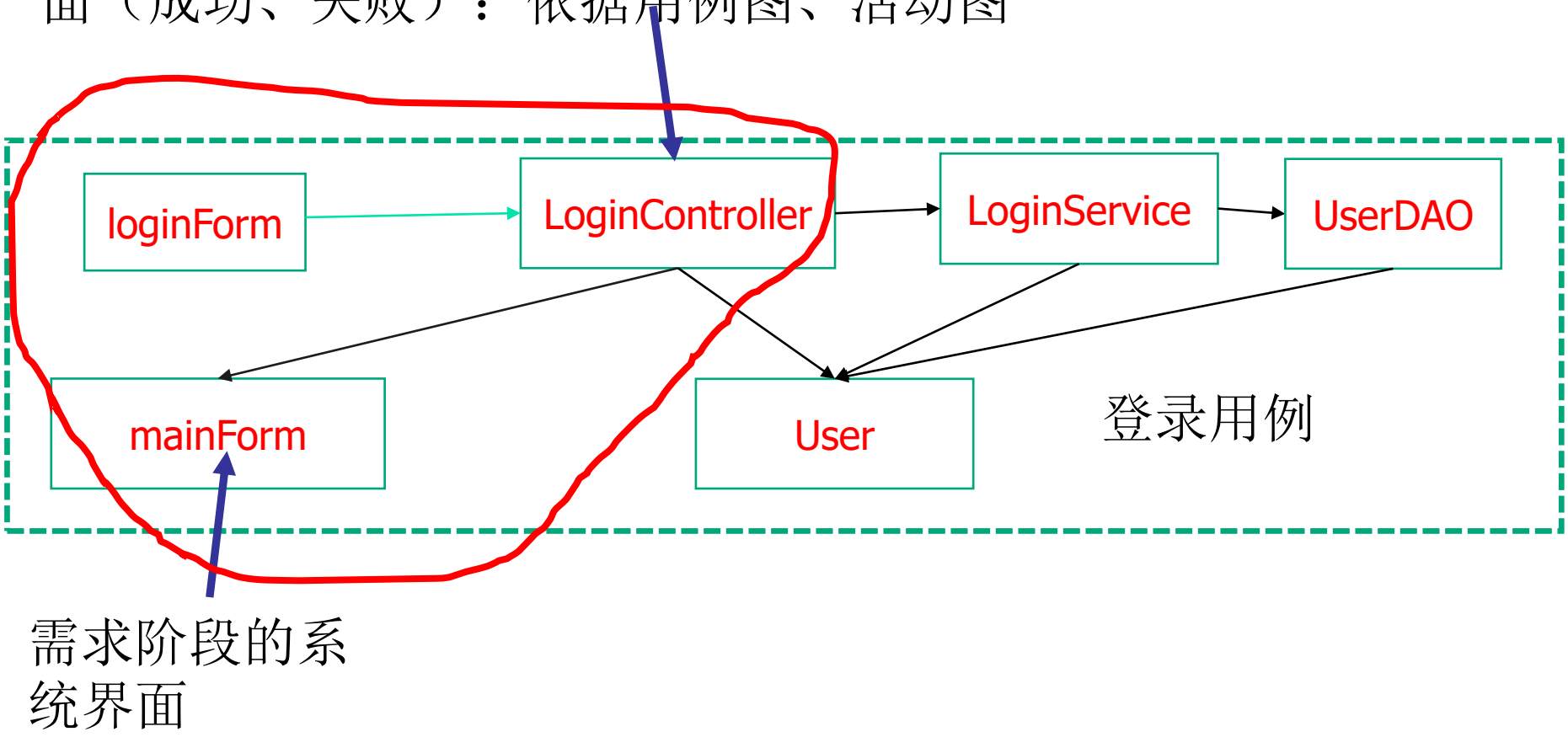
- 用户登录用例 (Android实现: 原生代码)



4.6.4 领域建模（类图）

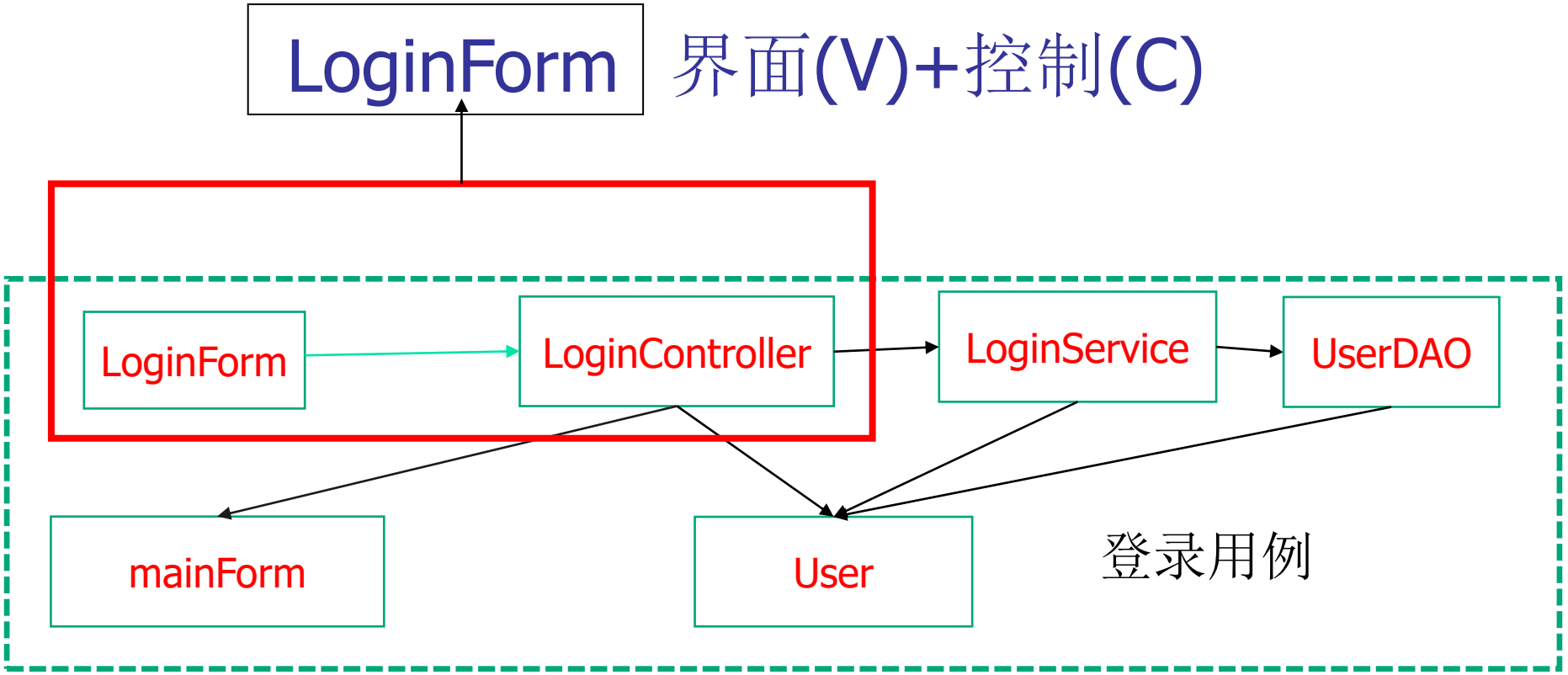
- 用户登录用例 (C/S实现1)

根据LoginService的结果返回给用户相应的页面（成功、失败）：依据用例图、活动图



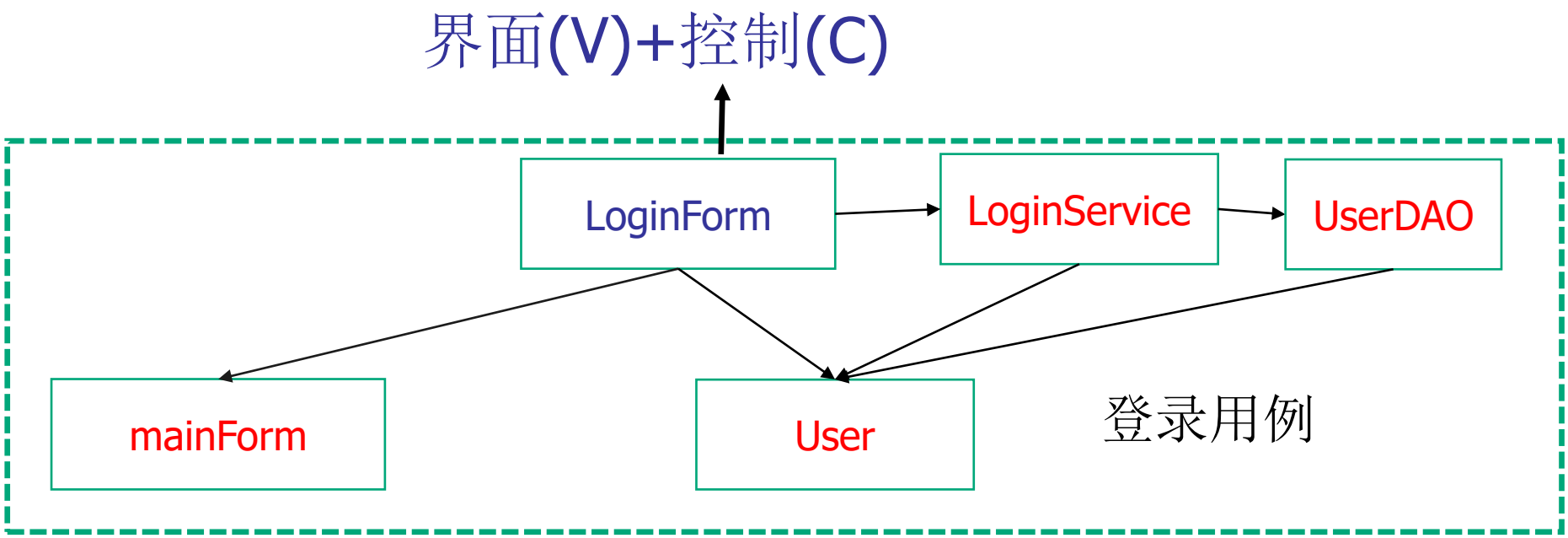
4.6.4 领域建模（类图）

- 用户登录用例 (C/S实现2)



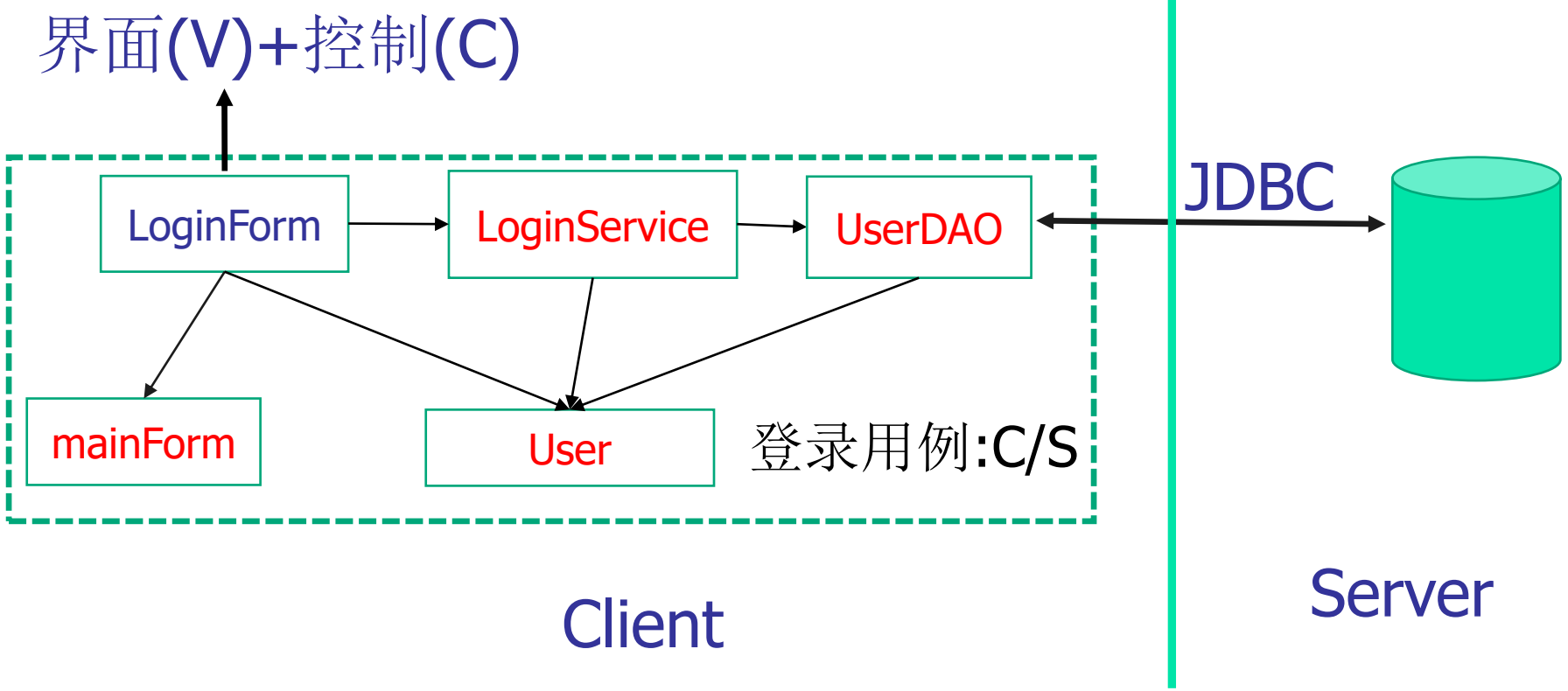
4.6.4 领域建模（类图）

- 用户登录用例 (C/S实现2)



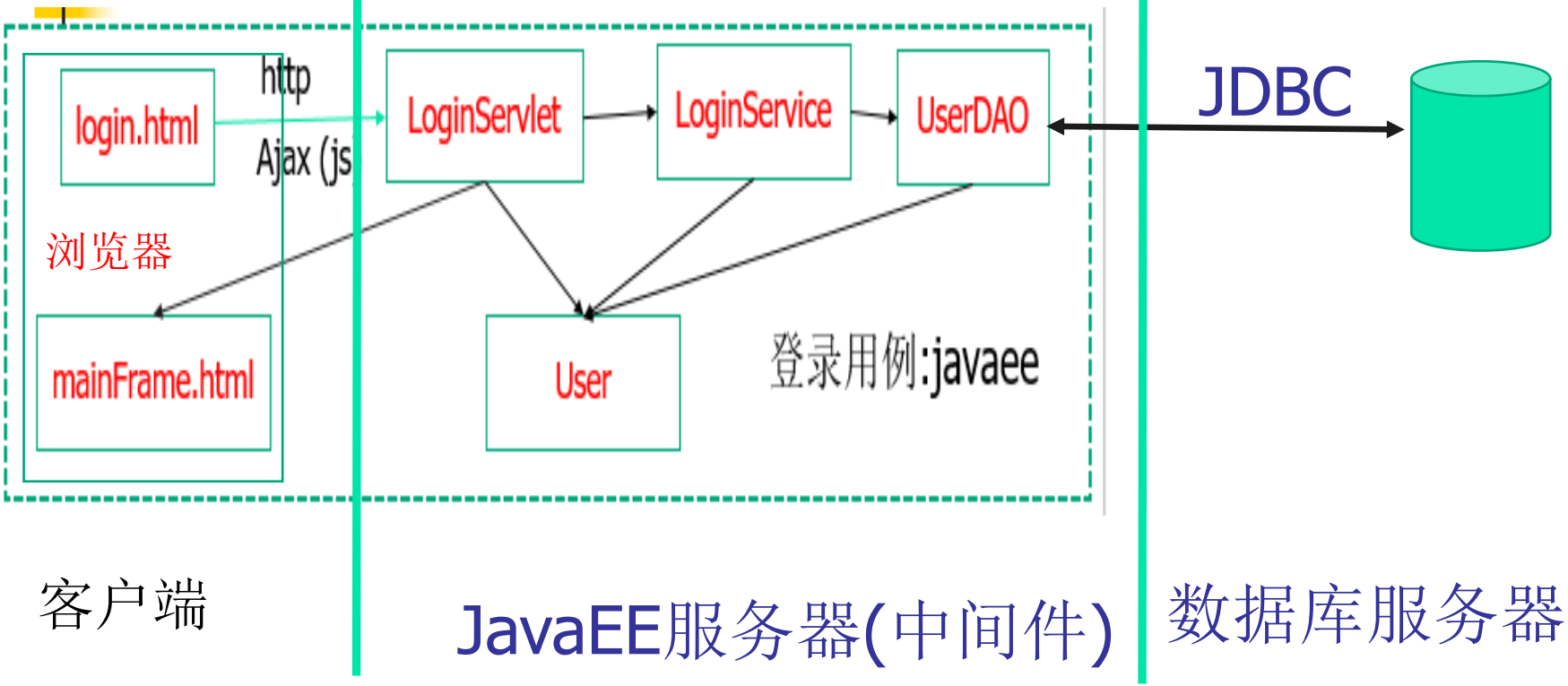
4.6.4 领域建模（类图）

- 用户登录用例 (C/S实现2)： 各类运行时所属机器



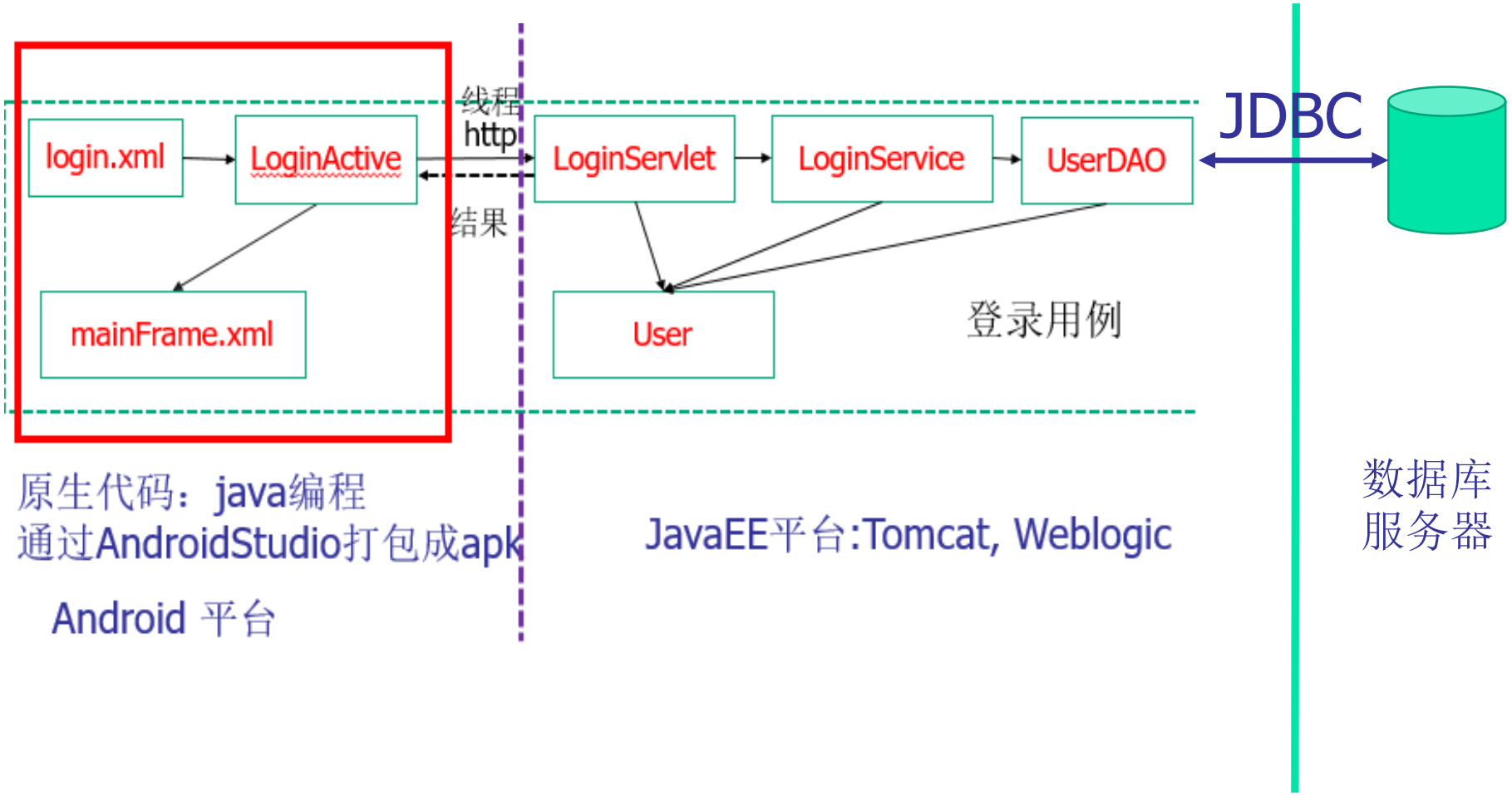
4.6.4 领域建模（类图）

- 用户登录用例 （JavaEE应用系统）： 各类运行时所属机器



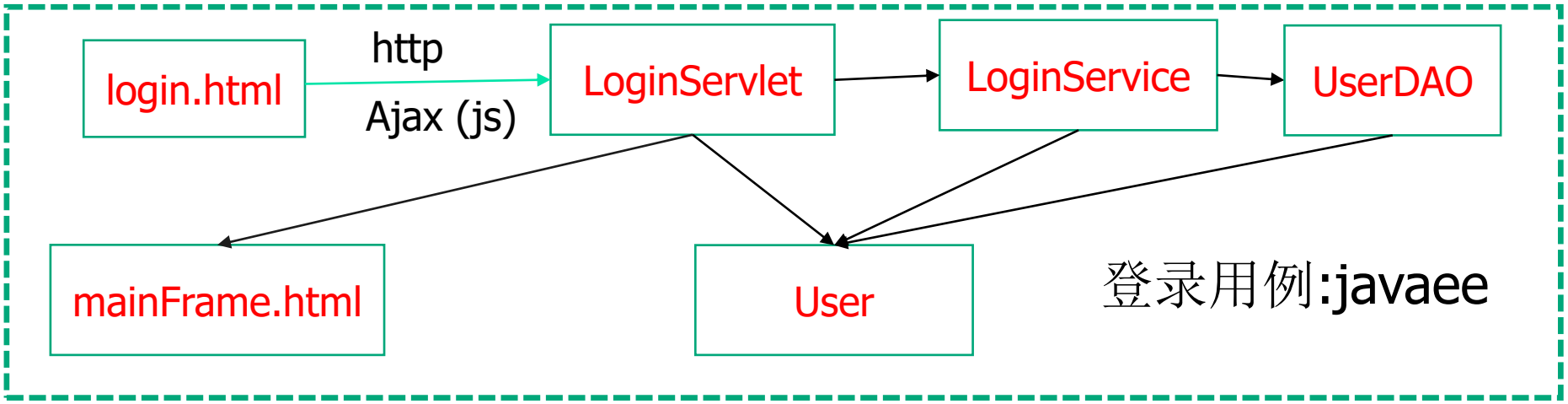
4.6.4 领域建模（类图）

- 用户登录用例 （ Android+JavaEE应用系统）：各类运行时所属机器



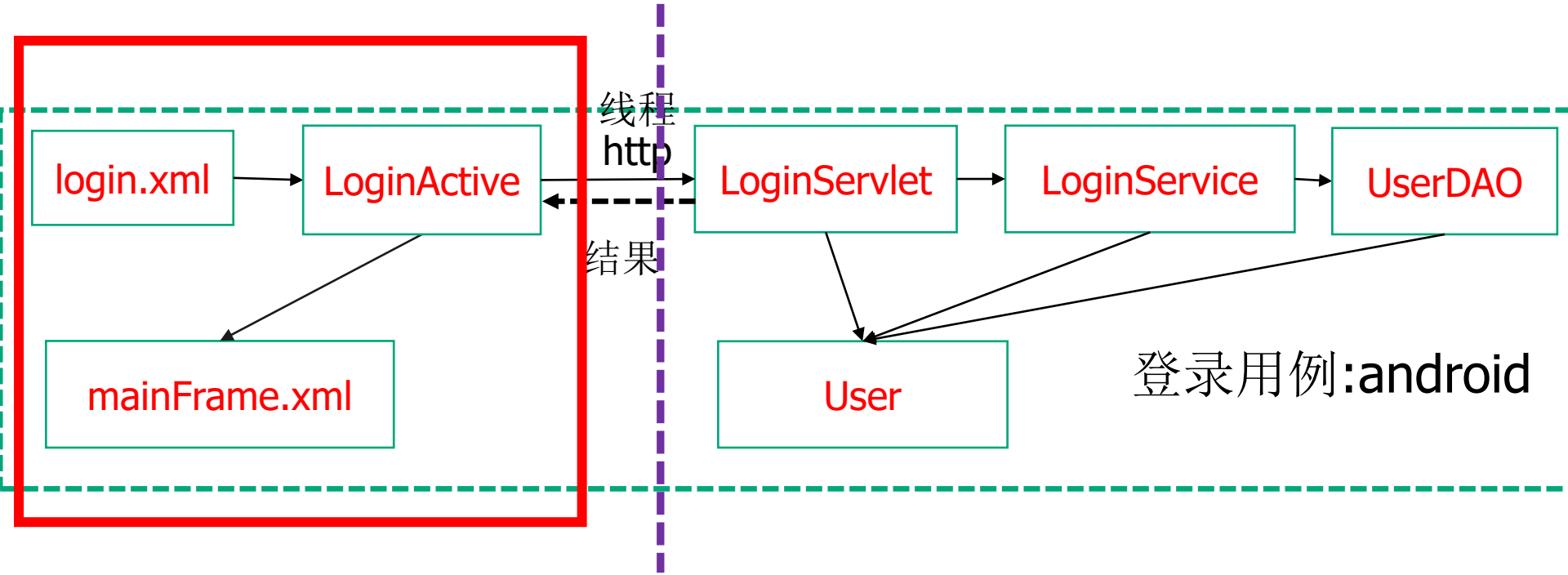
4.6.4 领域建模（类图）

- 用户登录用例软件设计（抽象类图）总结：
  三种主流软件体系结构风格



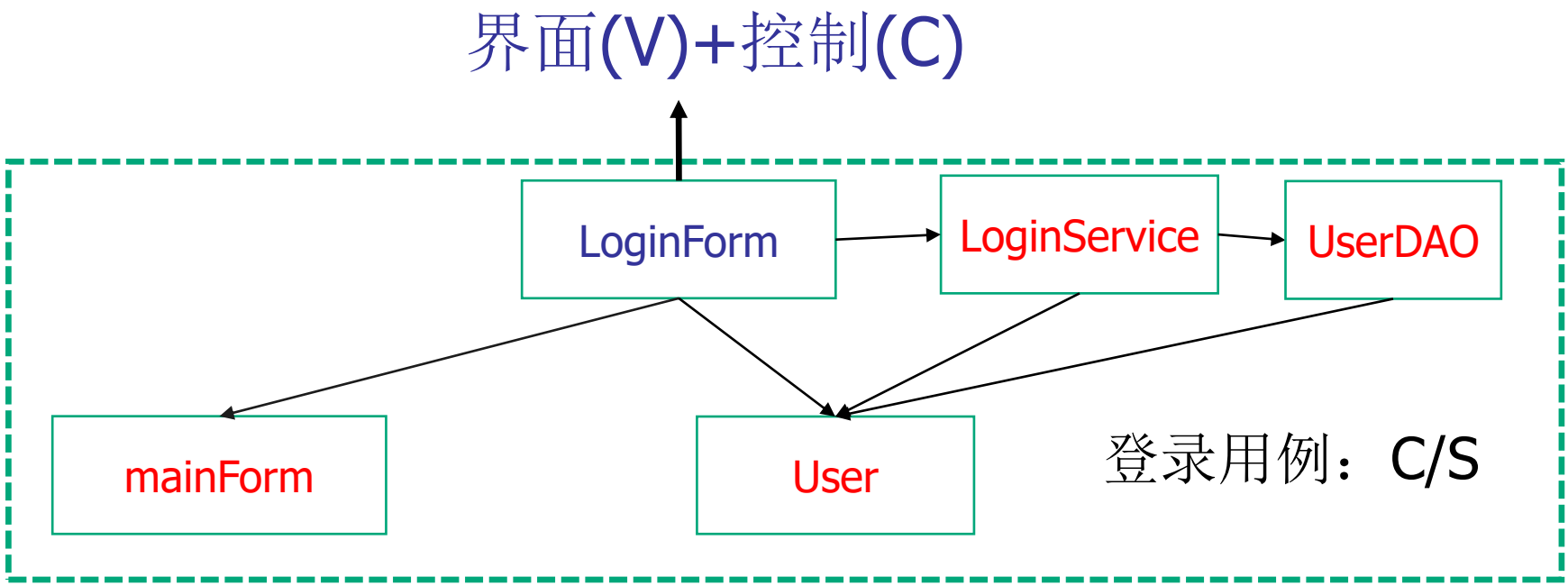
4.6.4 领域建模（类图）

- 用户登录用例软件设计（抽象类图）总结：
  三种主流软件体系结构风格



4.6.4 领域建模（类图）

- 用户登录用例软件设计（抽象类图）总结：
  三种主流软件体系结构风格



为什么要这样设计？

软件设计原则

抽象

体系结构

模式

模块化

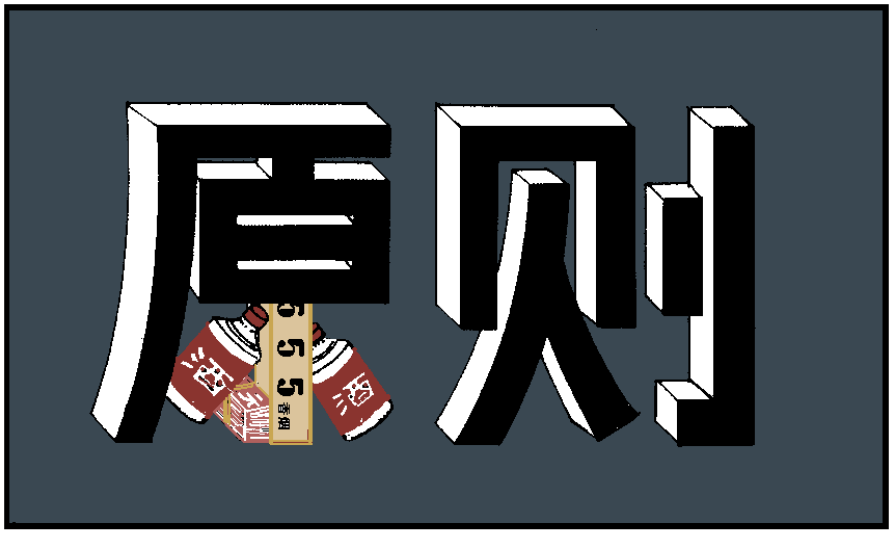
信息隐蔽

功能独立

求精

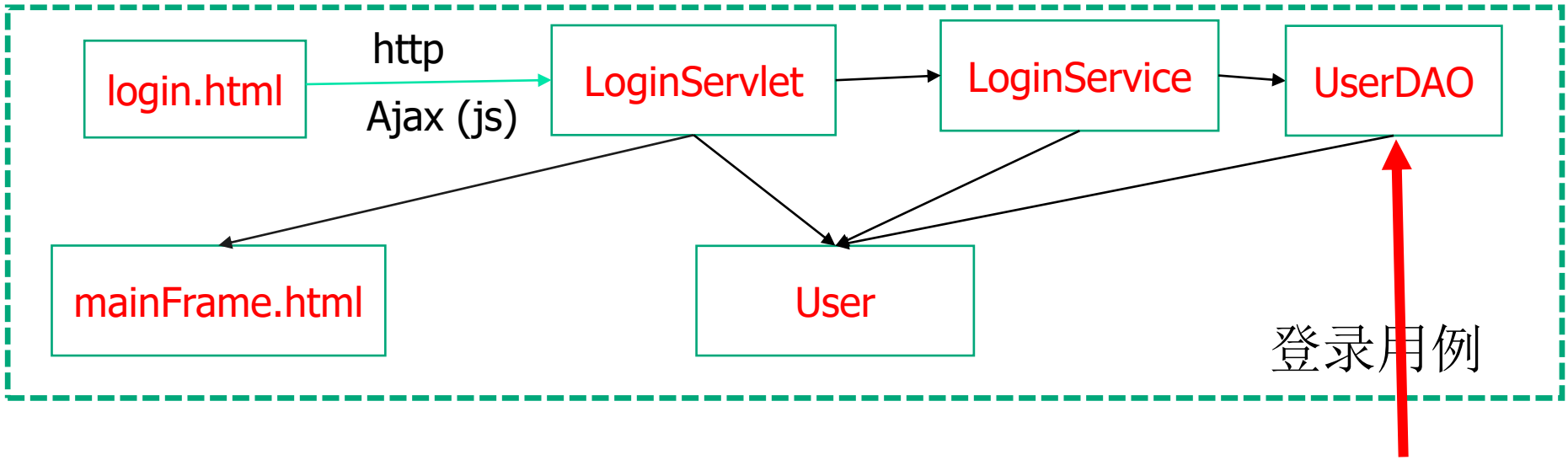
重用

设计类



原则下...

用户登录用例 (JavaEE实现)

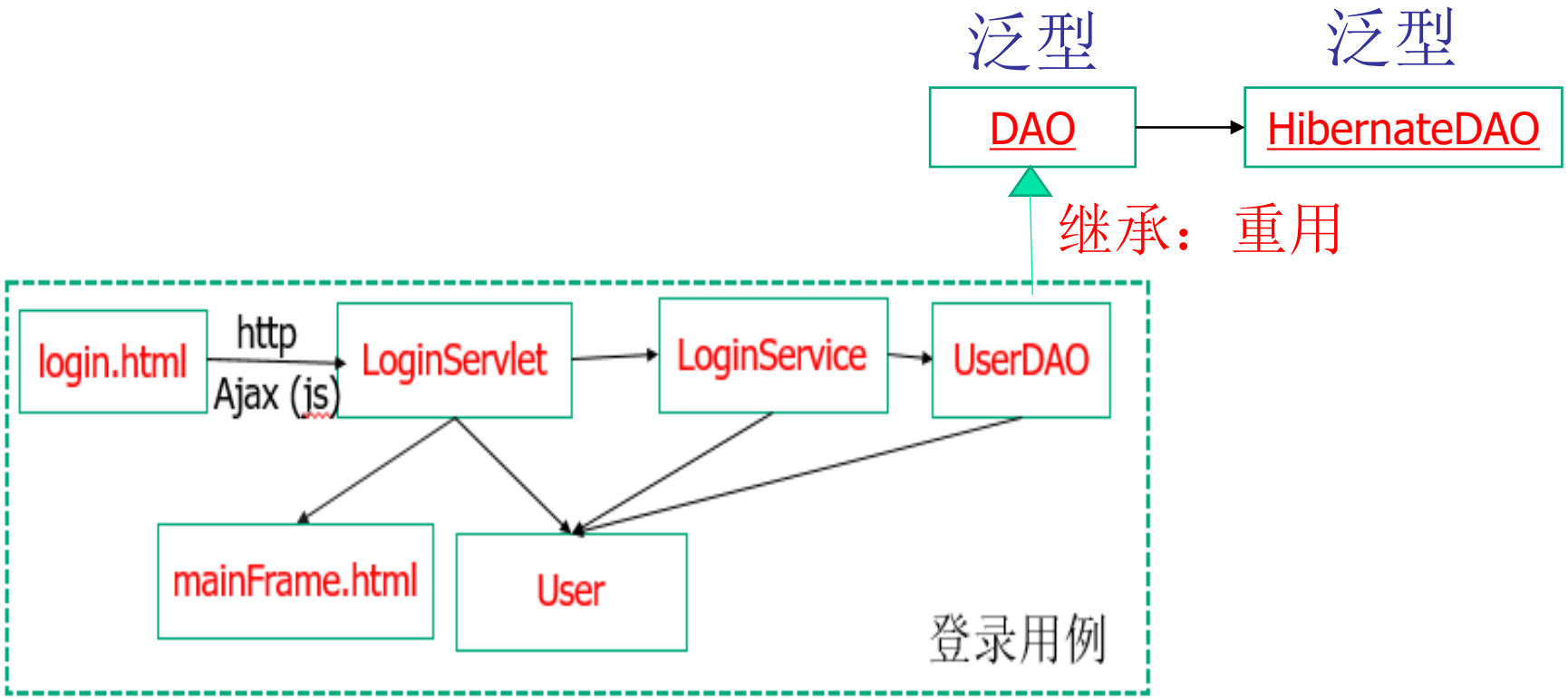


业务：调用存储过程

基本数据库操作：增、删、改、查询数据库操作。

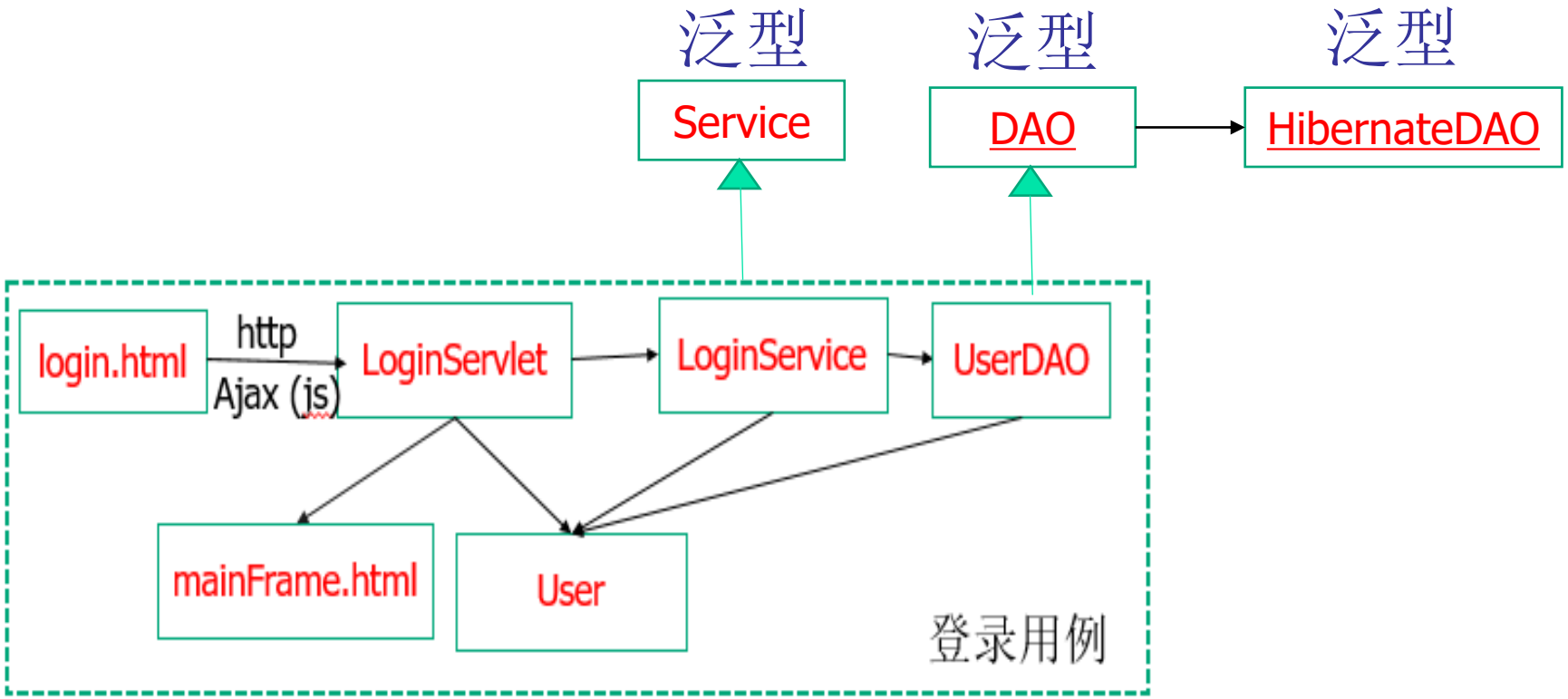
可以抽象出一个父类DAO: 实现基本的基本数据库操作

用户登录用例 (JavaEE实现)



优点：利用**重用技术**、泛型以及ORM(对象关系映射）技术，节省编写大量数据库操作代码。设计人员可以集中精力于业务流程。

用户登录用例 (JavaEE实现)



也可以抽象出一个Service父类（泛型)

thanks