Jakarta Struts CH06

出处 ：UML 软件工程组织

本章介绍 struts 应用程序的模型部分。模型表示的是应用程序的业务数据，模型与真实世界中的实体和业务过程紧密相关。我们将浏览模型组件在 Struts 框架中的角色及职责，还要为 storefront 程序创建一个体系结构合理、正确地业务层。此外，还特别描述关于持久化框架及其如何与 struts 应用程序集成的问题。

MVC 中所谓的 M

MVC 中的 M 代表“ 模型-视图-控制器 ”这一框架中的模型部分。对于使用软件系统的用户来说模型是软件制品中最有价值的部分。其中包括了业务实体和对于修改、访问业务实体的规则。特别重要的一点是，这些事务应该在有限的范围内完成，即上述那些与业务相关的代码应该比较集中且单一，以便维持应用程序的一致性、减少冗余和增强可复用性。

模型应该独立于用于访问业务对象的客户端技术。即框架中的模型部分不应该对上层（使用模型的其他层次）有任何的了解和假设。“ 依赖往下、数据往上 ”是一个总体的原则。也就是说，当设计和实现以各层次体系结构时，上层可以依赖于下层的特定实现，但下层不应该依赖于上层的特定实现。

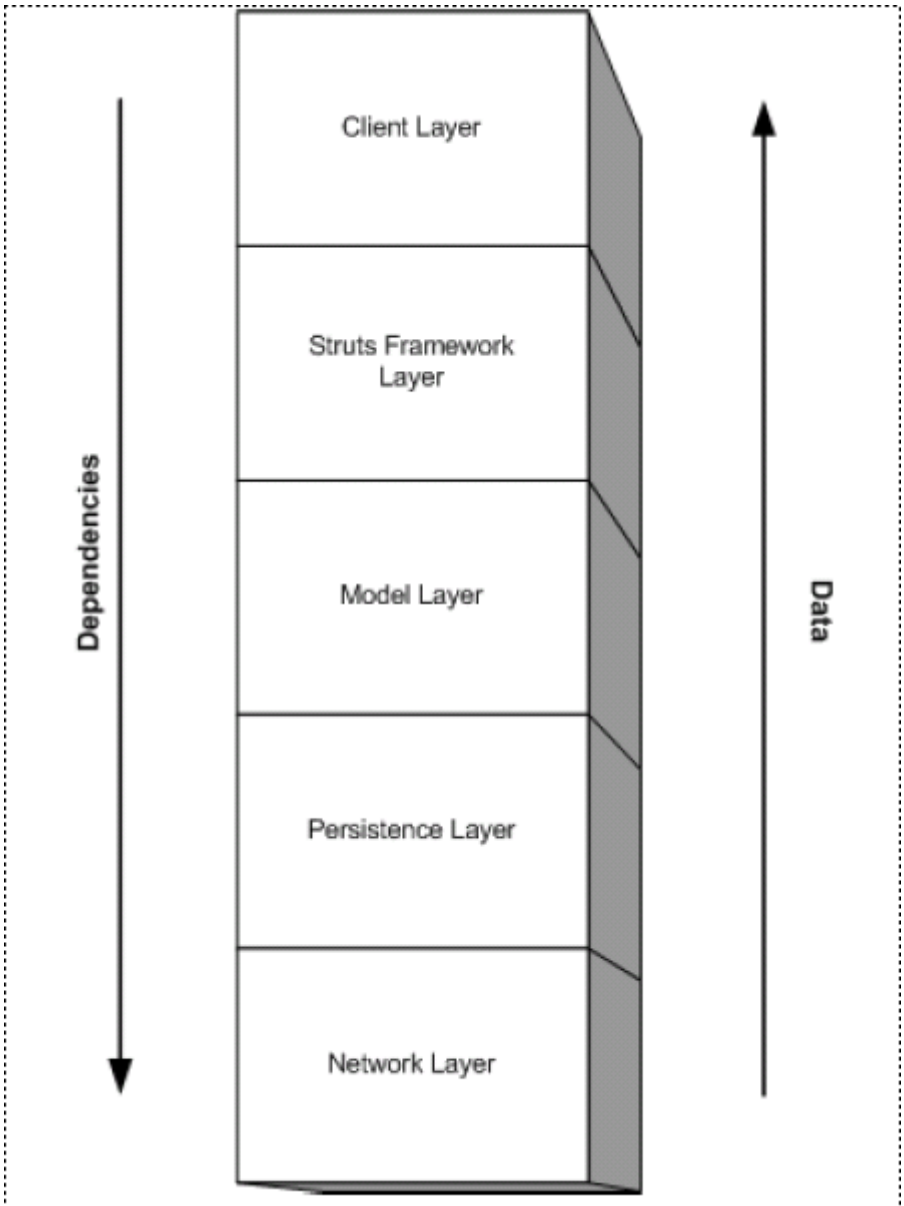


Figure 6-1.Application layers should only depend on lower layers

这里有一个非常简便的判断方法，如果在你的下层类中 import 或使用了上层的类。那么就可以说明你已近违反了这个原则。将上下层耦合在一起将使系统的维护、复用性、何可扩展性造成不利的影响。

在深入构建和设计 struts 应用程序的模型部分之前，首先要介绍一下模型的不同种类以及他们是如何共存在一格应用系统中的。

不同种类的模型

术语模型有很多不同的含义。通常是表示现实世界的某个方面。不管模型表示的是一个商店，还是一个拍卖行或是一个预告暴风雨的天气预报。所有这些例子都是现实世界中的真实概念或实体。创建模型的目的就是为了理解、描述和模拟真实世界。

在软件开发中术语模型指的是现实世界实体或概念的逻辑表示，在物理上也表现为成群的类和对象。首先应该做的是对问题领域进行全面的分析。一旦用例完成后，下一步就应该根据用例来开发出问题领域的概念模型。

开发概念模型

基于对问题领域的分析，根据问题空间的真实实体开发出概念模型。在概念模型中的实体是基于业务问题中的实际对象而非软件领域的部件。概念模型展示的是实际业务中的人、事、物、概念和他们的属性、他们之间的关系。而实体的行为则不包括在这种类型的模型中。

概念模型从系统的用例集中开发。其目的是帮助将实体标识出来以更好的理解问题领域。下图展示了 storefront 应用程序的概念模型。

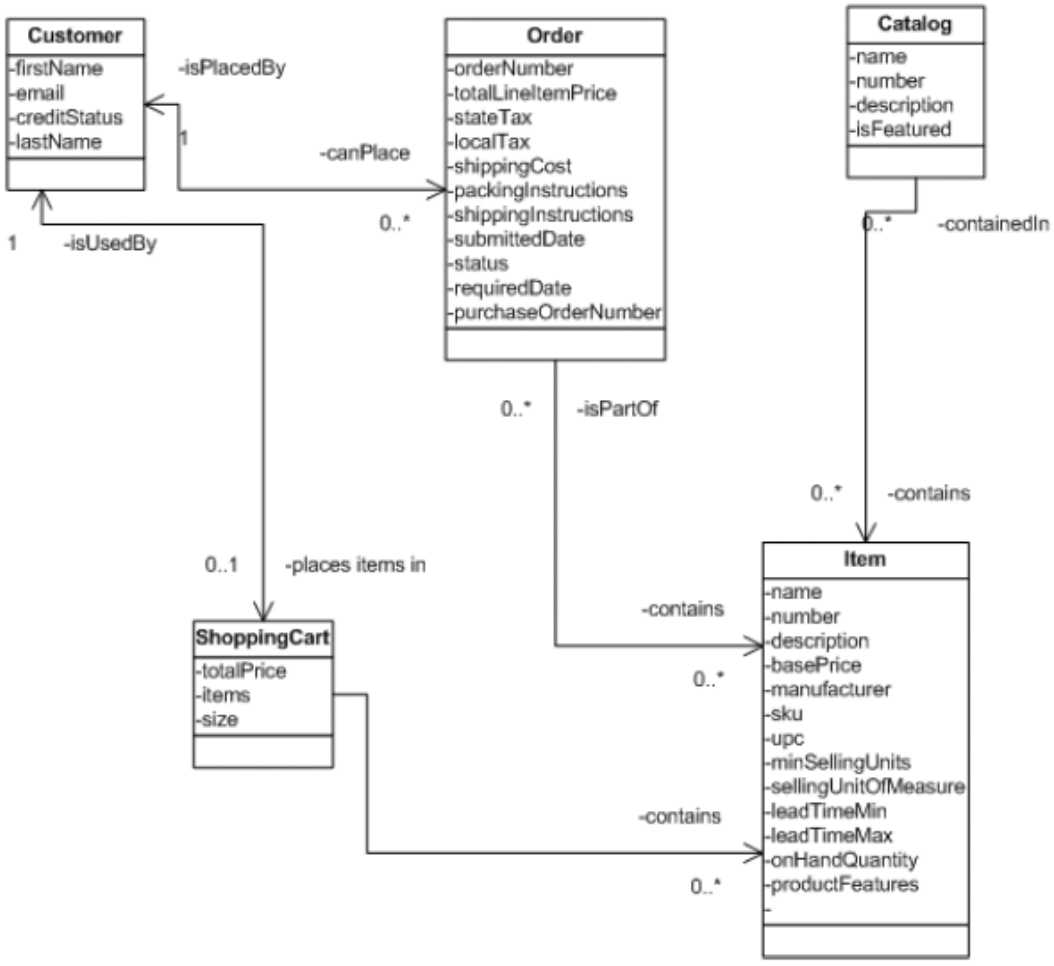


Figure 6-2. The Storefront Conceptual Model

应该注意到在上图中，实体的概念模型只包含了实体、属性及关系。在分析阶段并不指定实体属性

概念模型的价值在于其清晰地展现了活跃在问题领域的实体。每一个参与业务的人，无论是否是技术人员都应该能够看懂并理解概念模型。同时他们也应该能够快速地指出概念模型中含有的问题所在。然后就可以对概念模型进行调整。随后的设计和开发阶段需要依赖于这些分析阶段的制品，而在设计和开发阶段改变概念模型代价将是非常巨大的。

设计模型

概念模型只是分析阶段众多制品中的一项。在小型开发团队或小型项目中，当开发出概念模型后就可能直接跳入设计阶段。这里的风险是，当离开分析阶段时对业务领域问题是否已真正理解了。良好的理解需求及问题领域将使产品获得成功。

当开发出了概念模型后，仍然需要开发出对应的设计文档。这些设计文档通常包括类图、交互图或其他设计阶段制品。最小情况下，设计阶段制品应该包括应用程序业务对象类图。下图展示了 storefront 应用程序的设计类图，设计类图的开发是基于概念模型进行的。

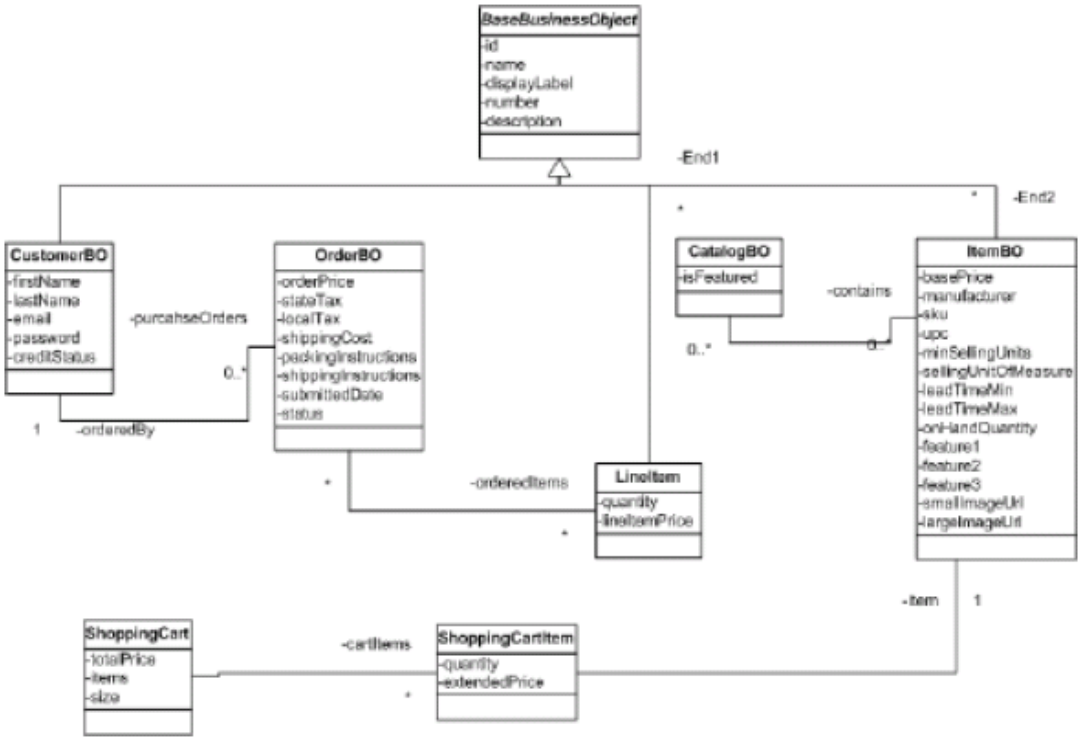


Figure 6-3. The Class Diagram for the Storefront Business Objects

为了简单起见上图并没有画出各个业务对象的方法集合，但是在这一阶段业务对象的方法集合已经被设计出来。

什么是业务对象

似乎并不能简单地给什么是业务对象这个问题下定义。通常“ 业务对象 ”有很多含义。每个开发者对这个术语也有着不同的理解。最简单来说业务对象就是对实际业务实体的软件抽象。其表达的是业务领域里的人、事、物、概念等等。比如在 storefront 中业务对象包括物品、订单、客户等等。

业务对象包含状态和行为。例如 OrderBO 对象，表示的是某个客户的订单，其中包含了价格、税率和订单状态。你也可以从订单业务对象中获取持有该订单的客户信息。业务对象的状态和行为是设计业务对象时最重要的问题。下面我们就具体研究一下业务对象的特性。

业务对象需要什么

一个业务对象需要满足以下条件：

- 包含状态和行为。
- 表达的是业务领域的人、事、物或概念。
- 必须是可复用的。

业务对象也可以被分成几种类型，典型的类型有：

- 业务实体对象
- 业务过程对象
- 业务事件对象

最常见的实体业务对象表达的是业务领域的人、事、物、场所或概念等等。他们可以直接从描述业务问题文档中的名词取得。在一个 EJB 应用程序中，业务实体对象被建模成 EntityBeans。在传统的 web 应用程序中他们通常用 JavaBeans 来表示含有行为及状态的业务对象。

业务过程对象表达的是应用程序中业务处理或工作任务流程。他们通常依赖于业务对象。并可以从描述业务问题文档中的动词部分提取。在一个 EJB 系统中，这类对象通常建模为 SessionBeans 或是消息驱动的 EJB，在非 EJB 应用程序中，他们通常由包含特定行为并在应用程序中成为管理器或控制器的 JavaBeans 表示。当然即便这些业务过程对象主要是实现业务处理或工作流程，但他们同样也可以保存应用程序的状态。以 EJB 来说，其就区分无状态的会话 Bean 和有状态的会话 Bean。

最后一类型的业务对象是业务事件对象。业务事件对象表达的是某个事件。例如：异常、通知、

时间等等。用户界面按动某个按钮也是触发一个事件。业务事件对象主要针对业务领域中的事件。

业务对象的重要性

在应用程序中使用业务对象具有很多好处。最重要的是他们提供了通用的术语和观点，这些术语和观点是通用的，因此人们可以使用统一的语言和观念进行工作。如果，有多个应用程序运行在同样的业务领域。那么业务对象层就可以在整个组织内部进行复用。

业务对象还能够随着组织业务的变化而能够轻松调整和修正。最后，业务对象具有定义良好的接口。内部具体的实现被封装，比如：假设你有一个使用 java.util.ArrayList 的业务对象。为了不使这个业务对象的类型暴露，就应该使用 java.util.List。此后如果内部实现由 ArrayList 改变成了 LinkedList。那么对于调用者来说就不需要进行任何修改。使用这些基类型或基础接口能够对系统作到很好的 ocp 原则。

持久化

通常，持久化的含义是应用程序中的数据能够持久保存。对数据的持久化是非常重要的，无路是小型应用还是大型应用。

持久化业务对象

当对象在应用程序的内存中被创建后，他们不可能永远留滞在那里。如果在该对象被清理之前需要需要保存这个对象的话就应该使用持久化技术，否则，当垃圾回收器将对象所占用的内存回收掉后，这个对象的所有状态将丢失。业务对象表达的信息有其需要被保存，比如订单、客户等信息。一旦信息被持久化保存后，那么程序就能很容易的重新获取这些数据。

将对象保存于关系模型

尽管保存数据的方法有很多，但关系数据库系统是最常用的方式。尽管如此，要有效使用关系数据库对对象进行持久化仍然必须克服一些障碍。其中最大的障碍是“ Impedance Mismatch ”。

Impedance Mismatch

对象保存状态和行为，并且能够与其他对象进行互操作。另一方面，关系模型是基于存储数据和联接数据为基础的。也就是说，关系模型是一种胖的数据视图。这里的挑战就是如何将程序世界的对象转换为关系模型世界的数据表示模型

详细的讨论如何将对象体系转变为关系模型超出了本书的范围。可以通过 *http://www.ambysoft.com/mappingObjects.pdf*

来访问到相关的资源。

你将很快就能看到，有很多 Object-to-Relational Mapping(ORM)框架使得 Java 开发者能够比较容易地完成这一任务。

Struts 框架为模型提供了什么

坦率的说，struts 框架并没有提供任何的关于构建模型组件的支持。已经存在很多这方面的框架用于处理业务对象。包括 EBJ、JDO 当然也可以使用 ORM 框架将 JavaBean 构造成业务对象。Struts 并不限制你使用何种手段去构造业务对象。

注：ActionForm 并不时业务对象

许多开发者在学习 ActionForm 类的时候通常会将其误认为是业务对象。尽管 ActionForm 符合业务对象特性的第一条准则，即其保存状态并及拥有行为，但是其内部的状态仅仅限于用户的输入。用户的输入是指从客户端上取得的数据，这些数据将被 ActionForm 对象保存，通过传输和验证传入业务对象中。

你已看到，在应用程序中使用业务对象的重要性。业务对象能够被持久化并含有应用程序的业务逻辑。他们应该被复用。这些业务对象的特点是 ActionForm 对象不具备的。ActionForm 的用途十分明确，就是应用于 Web 容器，因此不可能实现复用。

下一章将详细描述关于 ActionForms 的内容。ActionForms 被设计用于捕获 HTML 数据、进行表示层验证和提供将表示层数据向业务层传递的作用。

建立 storefront 模型部分

在讨论完 struts 应用程序的模型部分之后，我们就使用这些原则来建立 storefront 的业务对象（即模型部分）。很显然，这是一个假象的例子，因此其不可能完整地表达现实事件的电子商务模型。尽管如此，其仍然包含足够多的内容来使你进一步理解本章的内容。

访问关系数据库

Storefront 应用程序的状态持久化将使用关系数据库。或许你已了解，目前存在很多关系型数据库可供选择。在能够使用 storefront 业务模型之前，有很多工作需要完成，他们主要有：

- 创建业务对象
- 创建数据库

- 将业务对象映射到数据库
- 测试以确定业务对象能够被持久化到数据库

你可以看到，上述任务中并没有哪一项涉及到 struts 框架。这便使你的业务模型层不受任何客户端类型的限制、不受到任何框架的限制。也就是说，业务模型层对其他层次没有任何了解。

为了在 struts 框架发生改变时，使业务对象不受任何影响，我们将介绍 Business Delegate Pattern。Business Delegate 就好像是在客户端有业务层业务对象的代理一样。他隐藏了十几的业务服务。使用这种模式将最大限度的在客户端和业务模型之间解耦。

创建 storefront 业务对象

现在我们知道业务对象包含数据和行为。他们是数据库中一条或多条记录的虚拟表示。比如：在 storefront 例子中，OrderBO 对象表示了一个物理的订单。订单对象也应该包含那些保证订单数据有效的业务逻辑。

注：业务数据验证应该属于哪个层

在 struts 应用程序中，选择将输入的验证功能放置于何处是比较麻烦的事情。一方面，这看起来应该是由框架部分来解决，因为框架是用户数据被接收的第一站。因此似乎验证逻辑应放在框架中，具体来说可以放在表示层的 ActionForm 或控制层的 Action 中。这样做的问题是，这种验证就紧密地与框架耦合了起来，因此在复用业务对象时，验证逻辑就不能够被再使用了。因此，从业务对象与特定类型框架或客户端隔离的原理来说，对数据的验证部不应放于框架中（如果数据本身不用于业务层层而是被框架层使用，则另当别论）。

尽管业务逻辑不应该属于 struts 框架，但是有一种验证类型能够并且应该位于框架中。这种类型的验证通常称之为表示层验证。

表示层验证或称输入验证的目的基本上是三种：

- 词汇
- 语法
- 语义

词汇验证指的是检查输入的数据其形式是否符合要求。比如：输入的是否为一个整型数。语法验证以验证输入的合成类型的数据的形式是否正确。比如：时间日期型要求的格式是月/日/年。语

义验证指的是输入的数据格式是否正确，必须保证输入的格式必须正确且输入的值必须由意义。比如订单中订货的数量输入为-3，这就是没有意义的数字，尽管在词汇上和语法上其都正确。

表示层验证属于 struts 框架，但业务层验证则不属于框架。业务对象必须保证最终进入数据库的数据是有效、正确的，因此业务层有对数据进行验证的职责。

构建业务层的第一步是创建业务对象。在例子中我们只使用 JavaBean 来实现。因为许多业务对象都共享相同的几个属性，因此我们首先构造一个缺省的抽象基类：

```
abstract public class BaseBusinessObject
    implements java.io.Serializable {
    private Integer id;
    private String displayLabel;
    private String description;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getDescription() {
```

```
        return description;

    }

    public void setDisplayLabel(String label) {

        this.displayLabel = label;

    }

    public String getDisplayLabel() {

        return displayLabel;

    }

}
```

下面是 OrderBO 业务对象，其表示已各订单。

```
import java.math.BigDecimal;

import java.sql.Timestamp;

import java.util.Iterator;

import java.util.List;

import java.util.LinkedList;



public class OrderBO extends BaseBusinessObject {



    private List lineItems;
```

```
private CustomerBO customer;

private double totalPrice;

private Integer customerId;

private String orderStatus;

private Timestamp submittedDate;



public OrderBO() {

    super();

lineItems = new LinkedList();

}



public OrderBO( Integer id, Integer custId, String orderStatus,

    Timestamp submittedDate, double price ){

    this.setId(id);

    this.setCustomerId(custId);

    this.setOrderStatus(orderStatus);

    this.setSubmittedDate(submittedDate);

    this.setTotalPrice(price);

}
```

```
public void setCustomer( CustomerBO owner ){

    customer = owner;

}



public CustomerBO getCustomer(){

    return customer;

}



public double getTotalPrice(){

    return this.totalPrice;

}



private void setTotalPrice( double price ){

    this.totalPrice = price;

}



public void setLineItems( List lineItems ){

    this.lineItems = lineItems;

}
```

```
public List getLineItems(){

    return lineItems;

}



public void addLineItem( LineItemBO lineItem ){

    lineItems.add( lineItem );

}



public void removeLineItem( LineItemBO lineItem ){

    lineItems.remove( lineItem );

}



public void setCustomerId(Integer customerId) {

    this.customerId = customerId;

}



public Integer getCustomerId() {

    return customerId;

}
```

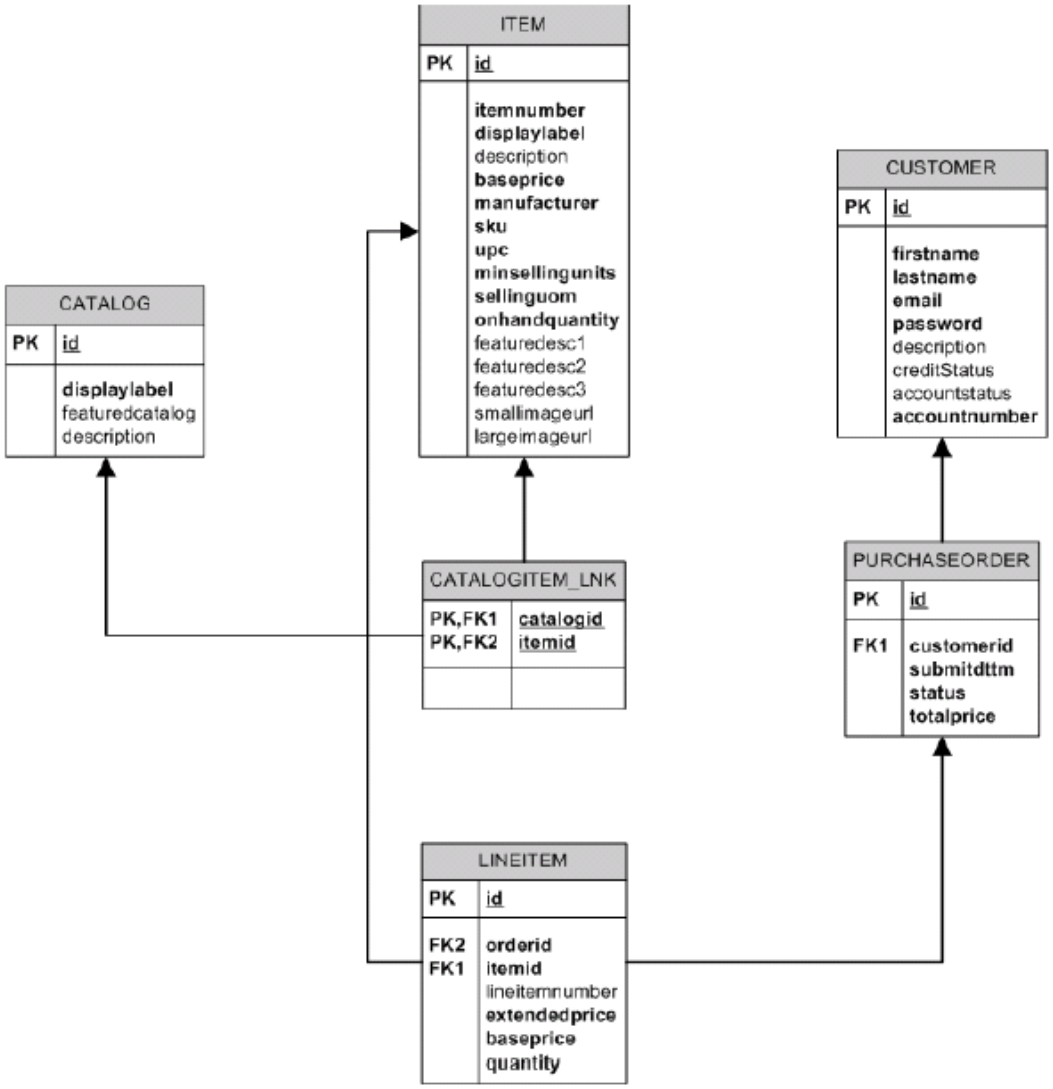
```
public void setOrderStatus(String orderStatus) {

    this.orderStatus = orderStatus;

}



public String getOrderStatus() {

    return orderStatus;

}



public void setSubmittedDate(Timestamp submittedDate) {

    this.submittedDate = submittedDate;

}



public Timestamp getSubmittedDate() {

    return submittedDate;

}



private void recalculatePrice(){

    double totalPrice = 0.0;

    if ( getLineItems() != null ){

        Iterator iter = getLineItems().iterator();
```

```
            while( iter.hasNext() ){
                Double lineItemPrice = ((LineItemBO)iter.next()).getUnitPrice();
                if (lineItemPrice == null){
                    return;
                }
                totalPrice += lineItemPrice.doubleValue();
            }
            setTotalPrice( totalPrice );
        }
    }
}
```

当设计你自己的业务对象时，不应该考虑他们将如何映射到数据库。也不需要因为害怕增加映射到数据库是会增加难度而不使用面向对象技术（如：继承和多态）。BaseBusinessObject 作为一个虚拟业务对象类就不会被映射到数据库中。

Storefront 数据模型

一旦数据对象构建完成后，就需要创建数据库模型（简称数据模型）。




```
# The SQL DDL for the Storefront Application

# (Apache Struts by O'Reilly)

# Chuck Cavaness

# Execute the next line if you need to clear the storefront database

DROP DATABASE storefront;

# Creates the initial database

CREATE DATABASE storefront;
```

```
# Make sure you are creating the tables in the storefront tablespace

use storefront;

CREATE TABLE CATALOG(

id int NOT NULL,

displaylabel varchar(50) NOT NULL,

featuredcatalog char(1) NULL,

description varchar(255) NULL

);

ALTER TABLE CATALOG ADD

CONSTRAINT PK_CATALOG PRIMARY KEY(id);

CREATE TABLE CUSTOMER (

id int NOT NULL ,

firstname varchar(50) NOT NULL,

lastname varchar(50) NOT NULL,

email varchar(50) NOT NULL ,

password varchar(15) NOT NULL,

description varchar(255) NULL,

creditStatus char(1) NULL,

accountstatus char(1) NULL,

accountnumber varchar(15) NOT NULL
```

```
);

ALTER TABLE CUSTOMER ADD

CONSTRAINT PK_CUSTOMER PRIMARY KEY(id);

CREATE TABLE ITEM (

id int NOT NULL ,

itemnumber varchar (255) NOT NULL ,

description varchar (255) NULL,

baseprice decimal(9,2) NOT NULL,

manufacturer varchar (255) NOT NULL,

sku varchar (255) NOT NULL,

upc varchar (255) NOT NULL,

minsellingunits int NOT NULL,

sellinguom varchar (255) NOT NULL,

onhandquantity int NOT NULL,

featuredesc1 varchar (255) NULL,

featuredesc2 varchar (255) NULL,

featuredesc3 varchar (255) NULL,

smallimageurl varchar (255) NULL,

largeimageurl varchar (255) NULL

)
```

```
ALTER TABLE ITEM ADD

CONSTRAINT PK_ITEM PRIMARY KEY(id);

CREATE TABLE CATALOGITEM_LNK(

catalogid int NOT NULL ,

itemid int NOT NULL

)

ALTER TABLE CATALOGITEM_LNK ADD

CONSTRAINT PK_CATALOGITEM_LNK PRIMARY KEY(catalogid, itemid);

ALTER TABLE CATALOGITEM_LNK ADD

CONSTRAINT FK_CATALOGITEM_LNK_CATALOG FOREIGN KEY

(catalogid) REFERENCES CATALOG(id);

ALTER TABLE CATALOGITEM_LNK ADD

CONSTRAINT FK_CATALOGITEM_LNK_ITEM FOREIGN KEY

(itemid) REFERENCES ITEM(id);

CREATE TABLE PURCHASEORDER (

id int NOT NULL,

customerid int NOT NULL,

submitdttm timestamp NOT NULL ,

status varchar (15) NOT NULL,

totalprice decimal(9,2) NOT NULL,
```

```
)

ALTER TABLE PURCHASEORDER ADD

CONSTRAINT PK_PURCHASEORDER PRIMARY KEY(id);

ALTER TABLE PURCHASEORDER ADD

CONSTRAINT FK_PURCHASEORDER_CUSTOMER FOREIGN KEY

(customerid) REFERENCES CUSTOMER(id);

CREATE TABLE LINEITEM (

id int NOT NULL,

orderid int NOT NULL,

itemid int NOT NULL,

lineitemnumber int NULL,

extendedprice decimal(9, 2) NOT NULL,

baseprice decimal(9, 2) NOT NULL,

quantity int NOT NULL

)

ALTER TABLE LINEITEM ADD

CONSTRAINT PK_LINEITEM PRIMARY KEY(id);

ALTER TABLE LINEITEM ADD

CONSTRAINT FK_LINEITEM_ORDER FOREIGN KEY

(orderid) REFERENCES PURCHASEORDER(id);
```

```
ALTER TABLE LINEITEM ADD

CONSTRAINT FK_LINEITEM_ITEM FOREIGN KEY

(itemid) REFERENCES ITEM(id);
```

在执行 DDL 以后，相应的数据库就建立完成了，你可以继续写一个 SQL 脚本用于插入些测试数据。

Object to Relational Mapping Framework(ORM)

现在就要执行创建业务层的第三步，将业务数据映射到数据库。完成这项工作可以由很多选择。你所做出的选择以来于几个因素：

- 使用 JDBC 调用
- 使用 ORM
- 使用私有的 Object-Relational Mapping 框架
- 使用对象数据库

总之，解决系统问题就是解决业务问题，不应该花费太多的时间在非核心的业务问题上。

目前有几个 ORM 产品可用，其中一些是商业性的也有开源软件：

Product	URL
TopLink	*http://www.objectwave.com/html/Main.html*
CocoBase	*http://www.cocobase.com*
Torque	*http://jakarta.apache.org/turbine/torque/index.html*
ObJectRelationalBridge	*http://objectbridge.sourceforge.net*
FrontierSuite	*http://www.objectfrontier.com*
Castor	*http://castor.exolab.org*

FreeFORM	*http://www.chimu.com/projects/form*
Expresso	*http://www.jcorporate.com*
JRelationalFramework	*http://jrf.sourceforge.net*
VBSF	*http://www.objectmatter.com*
Jgrinder	*http://sourceforge.net/projects/jgrinder*

无论你选择何种解决方案，你都必须保证这个方案的使用不汇延缓整个项目的进度。同时，你所选择的持久化层不应该被上层所依赖。这可以确保业务对象保持独立。//不同意，实际是数据模型是核心，业务实体类影射自数据表结构//如果你发现，你所选用的对象持久层框架要求你将其类库导入到业务对象层，那么这将是一个很严重的问题。一旦产生这样的问题，就应该使用Data Access Object(DAO)，以限制持久化访问框架对业务对象层的侵入。在 *http://www.object-relational.com/object-relational.html*.上你可以找到关于上述映射产品的比较。另外一个问题是，某些框架需要在业务数据对象被编译后改变其Java字节码，在使用这种框架之前必须保证你能掌握这种框架的使用。

Storefront 持久化框架

我们可以选择任何一种解决方案来实现 storefront 业务对象的持久化以将业务对象映射到数据库。我们的需求非常平常且业务模型也不复杂。对于选择持久化框架的几个要点如下：

- 这种框架机制对其它层特别是业务对象层的干扰有多大
- 解决方案的花费
- 可用的文献资料

费用问题是一个很到的因素。这个例子中我们选择了 ObjectRelationBrideg。他是一种开源框架且具有很多有价值的文档和资料。其使用 xml 文档来将业务数据映射到数据库中。Xml 文件将在运行时由框架进行解析。下面是一个典型的映射文件：

```
<ClassDescriptor id="120">

    <class.name>com.oreilly.struts.storefront.businessobjects.CustomerBO</class.name>
```

```
<table.name>CUSTOMER</table.name>

<FieldDescriptor id="1">

    <field.name>id</field.name>

    <column.name>id</column.name>

    <jdbc_type>INTEGER</jdbc_type>

    <PrimaryKey>true</PrimaryKey>

    <autoincrement>true</autoincrement>

</FieldDescriptor>

<FieldDescriptor id="2">

    <field.name>firstName</field.name>

    <column.name>firstname</column.name>

    <jdbc_type>VARCHAR</jdbc_type>

    </FieldDescriptor>

<FieldDescriptor id="3">

    <field.name>lastName</field.name>

    <column.name>lastname</column.name>

    <jdbc_type>VARCHAR</jdbc_type>

</FieldDescriptor>

<FieldDescriptor id="4">

    <field.name>email</field.name>
```

```
        <column.name>email</column.name>
        <jdbc_type>VARCHAR</jdbc_type>
    </FieldDescriptor>
    <FieldDescriptor id="5">
        <field.name>password</field.name>
        <column.name>password</column.name>
        <jdbc_type>VARCHAR</jdbc_type>
    </FieldDescriptor>
    <FieldDescriptor id="6">
        <field.name>accountStatus</field.name>
        <column.name>accountstatus</column.name>
        <jdbc_type>CHAR</jdbc_type>
    </FieldDescriptor>
    <FieldDescriptor id="7">
        <field.name>creditStatus</field.name>
        <column.name>creditstatus</column.name>
        <jdbc_type>CHAR</jdbc_type>
    </FieldDescriptor>
    <CollectionDescriptor id="1">
        <cdfield.name>submittedOrders</cdfield.name>
```

```
    <items.class>com.oreilly.struts.storefront.businessobjects.OrderBO</items.class>
        <inverse_fk_descriptor_ids>2</inverse_fk_descriptor_ids>
    </CollectionDescriptor>
</ClassDescriptor>
```

一旦所有的映射都做成 xml 文件后，你必须配置一个数据库联接信息、jdbc 驱动到 ObjectRelationalBridge 的配置文件中：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MappingRepository SYSTEM "repository.dtd" [
<!ENTITY user SYSTEM "repository_user.xml">
<!ENTITY junit SYSTEM "repository_junit.xml">
<!ENTITY internal SYSTEM "repository_internal.xml">
]>
<MappingRepository>
    <JdbcConnectionDescriptor id="default">
        <dbms.name>MsSQLServer2000</dbms.name>
        <jdbc.level>1.0</jdbc.level>
        <driver.name>com.ddtek.jdbc.sqlserver.SQLServerDriver</driver.name>
        <url.protocol>jdbc</url.protocol>
        <url.subprotocol>datadirect:sqlserver</url.subprotocol>
```

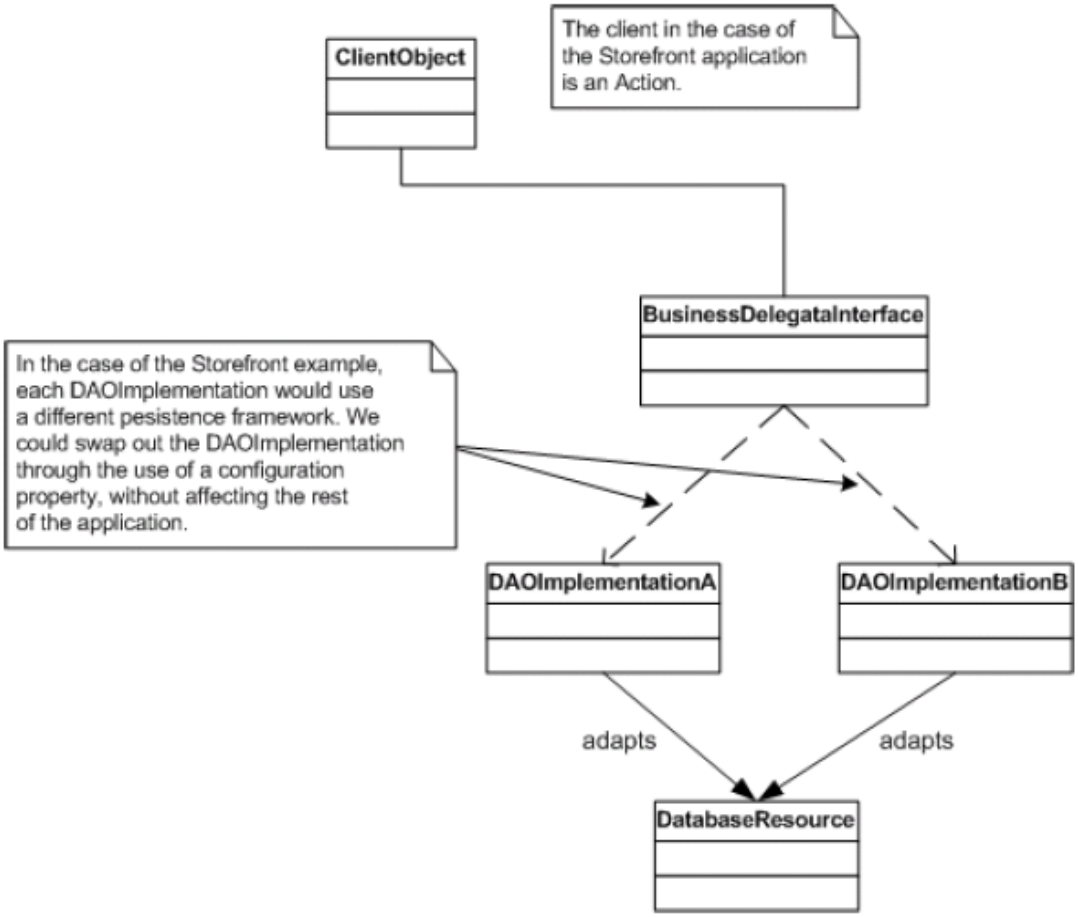
```
    <url.dbalias>//localhost:1433;DatabaseName=storefront;SelectMethod=cursor</url.dbalias>
        <user.name>sa</user.name>
        <user.passwd></user.passwd>
    </JdbcConnectionDescriptor>
<!-- include user defined mappings here -->
&user;
<!-- include ojb internal mappings here -->
&internal;
</MappingRepository>
```

BusinessDelegate 和数据访问对象（DAO）

最后还有一点问题需要解决，就是创建用于 storefront Action 类可使用的服务访问接口，这个接口能够使数据框架或业务对象与框架隔离开来，因此事实分重要的。这其中涉及到 DAO 模式的使用及 Bussiness Deleagete 模式的使用，首先我们介绍 DAO 模式。

DAO 模式的目的是将业务逻辑与数据访问逻辑隔离开来。在使用持久化框架时 DAO 对象也能够将持久化框架和业务对象层隔离开来。而且使持久化层能够被方便的替换。

实际上存在两种独立的设计模式包含在 DAO 模式中，他们是 Bridge 模式和 Adaptor 模式。在 storefront 应用程序中，我们将在 Action 和业务对象类中使用 DAO 和 Business Delegate 模式。



途中的客户对象是某个 Action 类。他们将获得服务接口即图中的 BusinessDelegateInterface。下面是这个接口的代码：

```
package com.oreilly.struts.storefront.service;

import java.util.List;

import com.oreilly.struts.storefront.catalog.view.ItemDetailView;

import com.oreilly.struts.storefront.catalog.view.ItemSummaryView;

import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;

import com.oreilly.struts.storefront.framework.security.IAuthentication;
```

```
public interface IStorefrontService

    extends IAuthentication {

    public List getFeaturedItems()

        throws DatastoreException;

    public ItemDetailView getItemDetailView( String itemId )

        throws DatastoreException;

}
```

DAOImplementationA 和 DAOImplementationB 是实现这个接口的两个数据访问对象，这两个对象实际与持久化层交互。持久化层的实现取决于所选择的持久化框架，这里我们选择的是 ObjectRelationalBridge。其实现如下：

```
package com.oreilly.struts.storefront.service;

import java.sql.Timestamp;

import java.util.List;

import java.util.ArrayList;

import com.oreilly.struts.storefront.catalog.view.ItemDetailView;

import com.oreilly.struts.storefront.catalog.view.ItemSummaryView;

import com.oreilly.struts.storefront.framework.security.IAuthentication;

import com.oreilly.struts.storefront.customer.view.UserView;

import com.oreilly.struts.storefront.businessobjects.*;

// Import the exceptions used
```

```
import com.oreilly.struts.storefront.framework.exceptions.DatastoreException;

import
com.oreilly.struts.storefront.framework.exceptions.InvalidLoginException;

import
com.oreilly.struts.storefront.framework.exceptions.ExpiredPasswordException;

import
com.oreilly.struts.storefront.framework.exceptions.AccountLockedException;

// Import the implementation specific packages

import org.odmg.*;

import ojb.odmg.*;



public class StorefrontServiceImpl

    implements IStorefrontService{

    // Implementation specific references

    Implementation odmg = null;

    Database db = null;

    /**

     * Create the service, which includes initializing the persistence

     * framework.

     */

    public StorefrontServiceImpl()
```

```
        throws DatastoreException {
        super();
        init();
    }
    /**
     * Return a list of items that are featured.
     */
    public List getFeaturedItems()
        throws DatastoreException {
        // Start a transaction
        Transaction tx = odmg.newTransaction();
        tx.begin();
        List results = null;
        try{
            OQLQuery query = odmg.newOQLQuery();
            // Set the OQL select statement
            query.create( "select featuredItems from " +
ItemBO.class.getName() );
            results = (List)query.execute();
            tx.commit();
```

```
        }catch( Exception ex ){

            / / Rollback the transaction

            tx.abort();

            ex.printStackTrace();

            throw DatastoreException.datastoreError(ex);

        }

        int size = results.size();

        List items = new ArrayList();

        for( int i = 0; i < size; i++ ){

            ItemBO itemBO = (ItemBO)results.get(i);

            ItemSummaryView newView = new ItemSummaryView();

            newView.setId( itemBO.getId().toString() );

            newView.setName( itemBO.getDisplayLabel() );

            newView.setUnitPrice( itemBO.getBasePrice() );

            newView.setSmallImageURL( itemBO.getSmallImageURL() );

            newView.setProductFeature( itemBO.getFeature1() );

            items.add( newView );

        }

        return items;

    }
```

```
/**
* Return an detailed view of an item based on the itemId argument.
*/
public ItemDetailView getItemDetailView( String itemId )
    throws DatastoreException{
    // Start a transaction
    Transaction tx = odmg.newTransaction();
    tx.begin();
    List results = null;
    try{
        OQLQuery query = odmg.newOQLQuery();
        // Set the OQL select statement
        String queryStr = "select item from " + ItemBO.class.getName();
        queryStr += " where id = $1";
        query.create(queryStr);
        query.bind(itemId);
        // Execute the transaction
        results = (List)query.execute();
        tx.commit();
    }catch( Exception ex ){
```

```
        // Rollback the transaction
        tx.abort();
        ex.printStackTrace();
        throw DatastoreException.datastoreError(ex);
    }
    //
    if (results.isEmpty() ){
        throw DatastoreException.objectNotFound();
    }
    ItemBO itemBO = (ItemBO)results.get(0);
    // Build a ValueObject for the Item
    ItemDetailView view = new ItemDetailView();
    view.setId( itemBO.getId().toString() );
    view.setDescription( itemBO.getDescription() );
    view.setLargeImageURL( itemBO.getLargeImageURL() );
    view.setName( itemBO.getDisplayLabel() );
    view.setProductFeature( itemBO.getFeature1() );
    view.setUnitPrice( itemBO.getBasePrice() );
    view.setTimeCreated( new Timestamp(System.currentTimeMillis() ));
    view.setModelNumber( itemBO.getModelNumber() );
```

```
        return view;
    }
    /**
    * Authenticate the user's credentials and either return a UserView for the
    * user or throw one of the security exceptions.
    */
    public UserView authenticate(String email, String password)
        throws InvalidLoginException,
               ExpiredPasswordException,
               AccountLockedException,
               DatastoreException {
        // Start a transaction
        Transaction tx = odmg.newTransaction();
        tx.begin();
        // Query the database for a user that matches the credentials
        List results = null;
        try{
            OQLQuery query = odmg.newOQLQuery();
            // Set the OQL select statement
            String queryStr = "select customer from " +
```

```
CustomerBO.class.getName();

            queryStr += " where email = $1 and password = $2";

            query.create(queryStr);

            // Bind the input parameters

            query.bind( email );

            query.bind( password );

            // Retrieve the results and commit the transaction

            results = (List)query.execute();

            tx.commit();

        }catch( Exception ex ){

            // Rollback the transaction

            tx.abort();

            ex.printStackTrace();

            throw DatastoreException.datastoreError(ex);

        }

        // If no results were found, must be an invalid login attempt

        if ( results.isEmpty() ){

            throw new InvalidLoginException();

        }

        // Should only be a single customer that matches the parameters
```

```
        CustomerBO customer = (CustomerBO)results.get(0);

        // Make sure the account is not locked

        String accountStatusCode = customer.getAccountStatus();

        if ( accountStatusCode != null && accountStatusCode.equals( "L" ) ){

            throw new AccountLockedException();

        }

        // Populate the Value Object from the Customer business object

        UserView userView = new UserView();

        userView.setId( customer.getId().toString() );

        userView.setFirstName( customer.getFirstName() );

        userView.setLastName( customer.getLastName() );

        userView.setEmailAddress( customer.getEmail() );

        userView.setCreditStatus( customer.getCreditStatus() );

        return userView;

    }

    /**

    * Log the user out of the system.

    */

    public void logout(String email){

        // Do nothing with right now, but might want to log it for auditing reasons
```

```
    }

    /**
    * Opens the database and prepares it for transactions
    */
    private void init()
        throws DatastoreException {
        // get odmg facade instance
        odmg = OJB.getInstance();
        db = odmg.newDatabase();
        //open database
        try{
            db.open("repository.xml", Database.OPEN_READ_WRITE);
        }catch( Exception ex ){
            throw DatastoreException.datastoreError(ex);
        }
    }
}
```

服务实现类提供了接口中所有规定方法的实现。因为 IstorefrontService 接口扩展了 Iauthentication 接口，所以 StorefrontServiceImpl 类必须实现 Iauthentication 接口规定的方法。注意，接口对于实现是毫不知情的，其不知道 struts 框架或是 web 容器之类的事情。这就使该层能够服用和被替换。

我们早先提到过需要调用 ObjectRelationalBridge 框架中的方法来访问数据库。因此映射 xml 文件必须事先进行配置。其中 init()是初始化方法。当实现接口的类被初始化时，就会调用 init()方法，此时，它将读入相关的 xml 文件并进行解析，做好访问数据库的准备。

这个构造方法需要被客户端调用，我们可以将其配置为 Struts 插件或是一个 servlet 过滤器或为 ActionServlet 创建一个 init()方法。

```
public void init() throws ServletException {

    // Make sure to always call the super's init() first

    super.init();

    // Attempt to initialize the persistence service

    try{

        // Create an instance of the service interface

        StorefrontServiceImpl serviceImpl = new StorefrontServiceImpl();

        // Store the service into application scope

        getServletContext().setAttribute(IConstants.SERVICE_INTERFACE_KEY,
serviceImpl);

    }catch( DatastoreException ex ){

        // If there's a problem initializing the service, disable the web app

        ex.printStackTrace();

        throw new UnavailableException( ex.getMessage() );

    }

}
```

最后一步需要展示的是我们如何在 Action 类中调用 Storefront 服务接口：

```
package com.oreilly.struts.storefront.security;

import java.util.Locale;

import javax.servlet.http.*;

import org.apache.struts.action.*;

import com.oreilly.struts.storefront.customer.view.UserView;

import com.oreilly.struts.storefront.framework.exceptions.BaseException;

import com.oreilly.struts.storefront.framework.UserContainer;

import com.oreilly.struts.storefront.framework.StorefrontBaseAction;

import com.oreilly.struts.storefront.framework.util.IConstants;

import com.oreilly.struts.storefront.service.IStorefrontService;

/**

* Implements the logic to authenticate a user for the storefront application.

*/

public class LoginAction

    extends StorefrontBaseAction {

    /**

     * Called by the controller when the a user attempts to login to the

     * storefront application.
```

```
 * /

public ActionForward execute( ActionMapping mapping,

    ActionForm form,

    HttpServletRequest request,

    HttpServletResponse response )

    throws Exception{

    / / Get the user's login name and password. They should have already

    / / validated by the ActionForm.

    String email = ((LoginForm)form).getEmail();

    String password = ((LoginForm)form).getPassword();

    / / Login through the security service

    IStorefrontService serviceImpl = getStorefrontService();

    UserView userView = serviceImpl.authenticate(email, password);

    UserContainer existingContainer = null;

    HttpSession session = request.getSession(false);

    if ( session != null ){

        existingContainer = getUserContainer(request);

        session.invalidate();

    }else{

        existingContainer = new UserContainer();
```

```
        }

        // Create a new session for the user

        session = request.getSession(true);

        existingContainer.setUserView(userView);

        session.setAttribute(IConstants.USER_CONTAINER_KEY,
existingContainer);

        return mapping.findForward(IConstants.SUCCESS_KEY);

    }

}
```

getStorefrontService()方法是从 Action 基类中实现的，因为每个 Action 类奖需要调用这个方法。getStorefrontService()方法将返回 StorefrontServiceImpl 对象，该对象是在 ActionServlet 的 init()方法中创建的。

包含 getStorefrontService()方法的 StorefrontBaseAction 类如下：

```
package com.oreilly.struts.storefront.framework;

import java.util.Collection;

import java.util.LinkedList;

import java.util.List;

import java.util.Locale;

import java.util.Iterator;

import javax.servlet.http.*;
```

```
import org.apache.struts.action.*;

import com.oreilly.struts.storefront.framework.util.IConstants;

import com.oreilly.struts.storefront.framework.exceptions.*;

import com.oreilly.struts.storefront.service.IStorefrontService;

/**

* An abstract Action class that all store front action classes should

* extend.

*/

abstract public class StorefrontBaseAction

    extends Action {

    protected IStorefrontService getStorefrontService(){

        return
(IStorefrontService)getApplicationObject(IConstants.SERVICE_INTERFACE_KEY );

    }

    /**

     * Retrieve a session object based on the request and the attribute name.

     */

    protected Object getSessionObject( HttpServletRequest req,

        String attrName) {

        Object sessionObj = null;
```

```
        // Don't create a session if one isn't already present
        HttpSession session = req.getSession(true);
        sessionObj = session.getAttribute(attrName);
        return sessionObj;
    }
    protected String getLoginToken(HttpSession session) {
        return (String)session.getAttribute(IConstants.LOGIN_TOKEN_KEY);
    }
    public boolean isLoggedIn( HttpServletRequest request ){
        UserContainer container = this.getUserContainer(request);
        return ( container != null && container.getUserView() != null );
    }
    /**
    * Return the instance of the ApplicationContainer object.
    */
    protected ApplicationContainer getApplicationContainer() {
        return
(ApplicationContainer)getApplicationObject(IConstants.APPLICATION_CONTAINER_KEY);
    }
    protected void removeLoginToken(HttpSession session) {
```

```
        session.removeAttribute(IConstants.LOGIN_TOKEN_KEY);

    }

    /**

    * Retrieve the UserContainer for the user tier to the request.

    */

    protected UserContainer getUserContainer(HttpServletRequest request) {

        UserContainer userContainer = (UserContainer)getSessionObject(request,
IConstants.USER_CONTAINER_KEY);

        // Create a UserContainer for the user if it doesn't exist already

        if(userContainer == null) {

            userContainer = new UserContainer();

            userContainer.setLocale(request.getLocale());

            HttpSession session = request.getSession();

            session.setAttribute(IConstants.USER_CONTAINER_KEY, userContainer);

        }

        return userContainer;

    }

    /**

    * Retrieve an object from the application scope by its name. This is

    * a convience method.
```

```
     * /

    protected Object getApplicationObject(String attrName) {

        return servlet.getServletContext().getAttribute(attrName);

    }

    protected void setLoginToken( HttpSession session,

        String path) {

        session.setAttribute(IConstants.LOGIN_TOKEN_KEY, path);

    }

}
```

最后，我们注意到上述某些方法中使用到了值对象 UserView。

注意，Action 类的程序依赖于 IStorefrontService 接口，而非实际实现接口的对象。如果我们需要改变持久化框架那么只是要实现 IstorefrontService 即可，同时在 init()方法中替换掉实际服务对象的类名，更可取的做法是在 web.xml 文件中指定类的全名称，这样，在 actionServlet 中队服务实现对象也完全具有低耦合度了。

译者简介

FreeJet 对软件技术抱有浓厚兴趣的自学者，目前正跋涉于 OOAD 和软件工程过程的知识海洋。坚信软件系统应靠推导和演化得到。愿意与各位朋友交换苹果，分享 OO 这一宝贵的精神财富。可以通过 Freejet2k@hotmail.com 与我取得联系。