厦门大学《面向对象分析与设计》课程试卷

信息学院软件工程系 2000 年级软件工程专业

一、 简答题（每题 10 分，共 30 分）

1、 结合课程设计中的实际运用，简述层次体系结构中 Dao 层和 Service 层的职责是什么。

2、 简述面向功能设计、面向对象设计和函数式编程的各自特点是什么。

3、 简述领域模型和设计类图的相同点和区别。

二、 画图题（30 分）

图 1 是支付模块中建立退款的交易的相关代码，请根据代码画出相应的设计类图和时序图，要求除以下内容外，图 1 中的其余内容均需要画出。

1、 类图中可以省略属性的类型

2、 类图中可以省略方法的参数类型及返回类型

三、 分析题（40 分）：

图 2-图 14 的设计类图、时序图和代码描述的是 Product 获取关联的 Onsale 对象的设计。

1、 请简述该设计的目标。（10 分）

2、 运用 GRASP 方法和软件设计原则分析该设计目标是如何实现的。（30 分）

```
@Service
public class RefundService {
    private DivRefundTransDao divRefundTransDao;
    private ShopChannelDao shopChannelDao;
    private PayTransDao payTransDao;
    private RefundTransDao refundTransDao;
    private PayAdaptorFactory factory;
    …
    @Transactional
    public RefundTransDto createRefund(Long shopId, Long paymentId, Long amount, long divAmount,
                                       UserDto user) {
        //获得支付交易
        PayTrans payTrans = this.payTransDao.findById(paymentId);
        //获得支付交易的商铺渠道属性
        ShopChannel shopChannel = payTrans.getShopChannel();
        //通过商铺渠道的商铺属性判断支付交易是否是同一商铺，
        if (PLATFORM != shopId && shopId != shopChannel.getShopId()) {
            throw new BusinessException(ReturnNo.RESOURCE_ID_OUTSCOPE,
                    String.format(ReturnNo.RESOURCE_ID_OUTSCOPE.getMessage(),
                      "退款对应的支付交易", paymentId, shopId));
        }
        //创建退款交易
        RefundTrans refundTrans = new RefundTrans(shopChannel, payTrans, amount, divAmount);
        this.refundTransDao.insert(refundTrans, user);
        PayAdaptor payAdaptor = this.factory.createPayAdaptor(shopChannel);
        //根据 payTrans 的 status 属性判断是否分账
        if (PayTrans.DIV == payTrans.getStatus()) {
            //获得支付交易的分账属性
            DivPayTrans divPayTrans = payTrans.getDivTrans();
            //需要先调用分账回退 API
            DivRefundTrans divRefundTrans = new DivRefundTrans(refundTrans, divPayTrans, shopChannel);
            this.divRefundTransDao.insert(divRefundTrans, user);
             this.payAdaptor.createDivRefund(refundTrans, divRefundTrans);
        }
        this.payAdaptor.createRefund(refundTrans);
        RefundTransDto dto = this.getDto(refundTrans, shopChannel);
        return dto;
    }
    …
}
```

图 1：退款交易代码

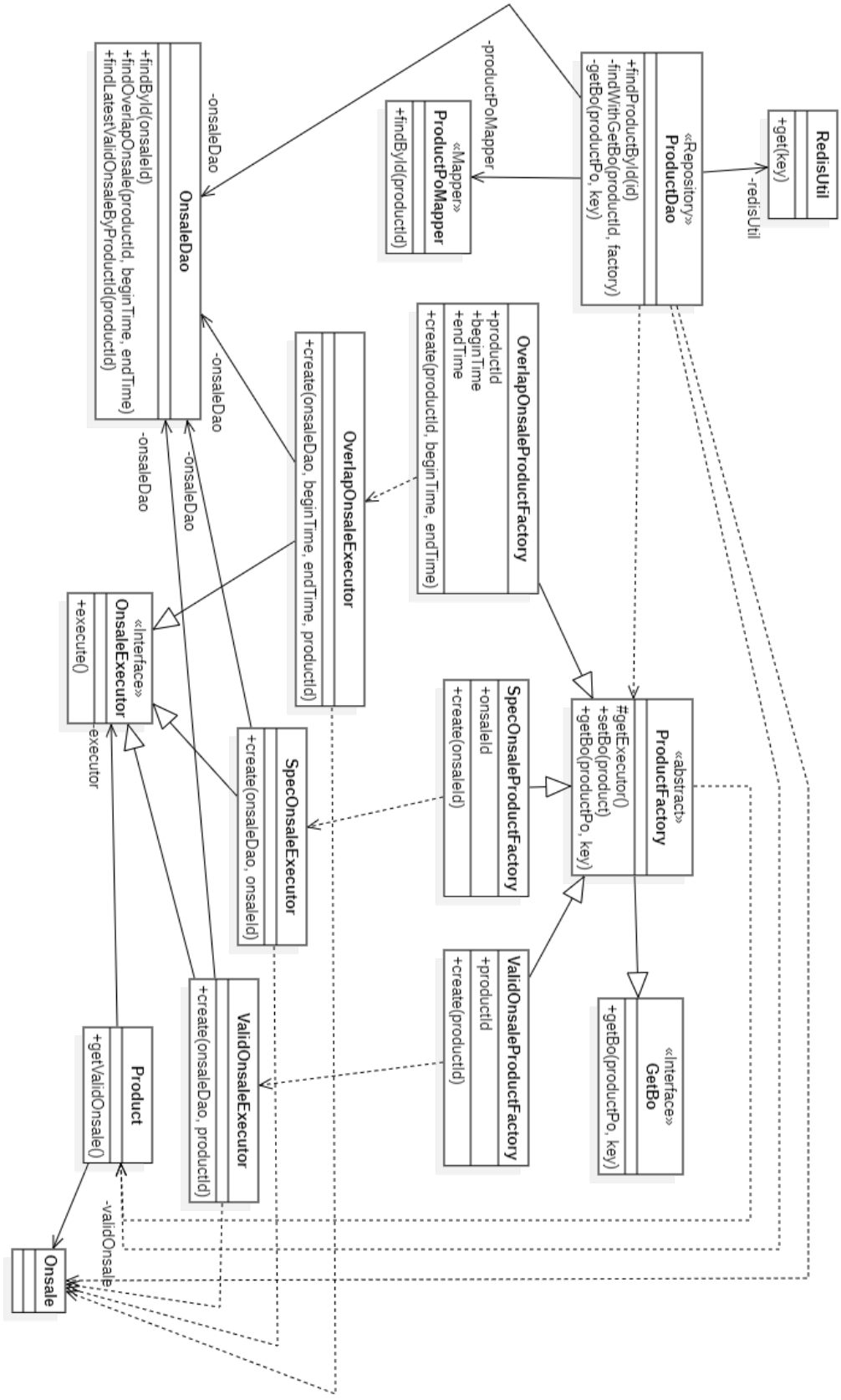
RedisUtil
+get(key)
-redisUtil
«Repository»
ProductDao
+findProductById(id)
-findWithGetBo(productId, factory)
-getBo(productPo, key)
-productPoMapper
«Mapper»
ProductPoMapper
+findById(productId)
-onsaleDao
OnsaleDao
+findById(onsaleId)
+findOverlapOnsale(productId, beginTime, endTime)
+findLatestValidOnsaleByProductId(productId)
OverlapOnsaleProductFactory
+productId
+beginTime
+endTime
+create(productId, beginTime, endTime)
OverlapOnsaleExecutor
+create(onsaleDao, beginTime, endTime, productId)
«Interface»
OnsaleExecutor
+execute()
executor
SpecOnsaleProductFactory
+onsaleId
+create(onsaleId)
«abstract»
ProductFactory
#getExecutor()
+setBo(product)
+getBo(productPo, key)
SpecOnsaleExecutor
+create(onsaleDao, onsaleId)
ValidOnsaleProductFactory
+productId
+create(productId)
«Interface»
GetBo
+getBo(productPo, key)
ValidOnsaleExecutor
+create(onsaleDao, productId)
Product
+getValidOnsale()
-validOnsale
Onsale

图 2：设计类图

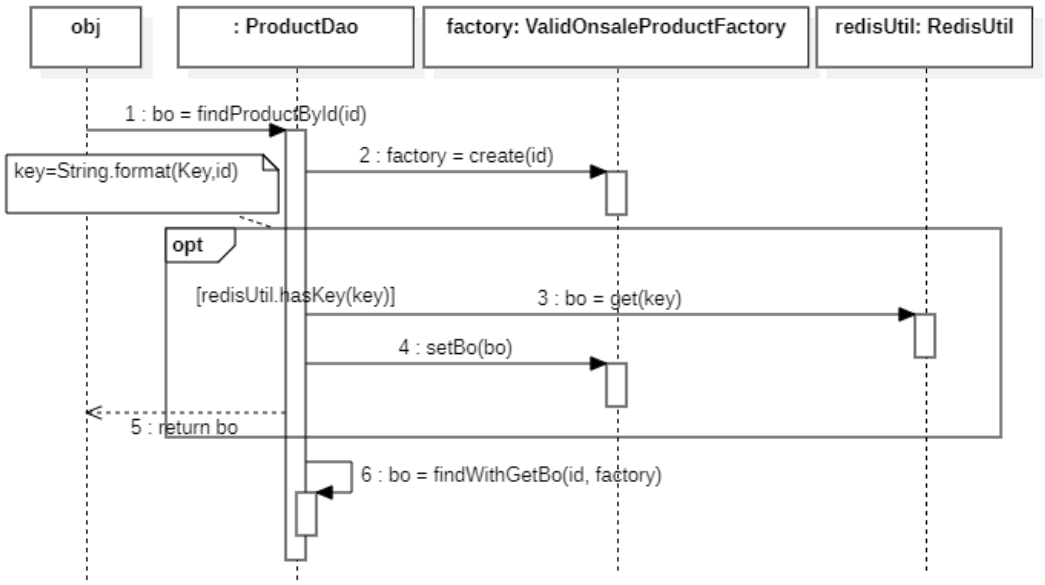

图 3：根据 productId 获得 Product 的对象模型，关联当前有效的 Onsale 对象

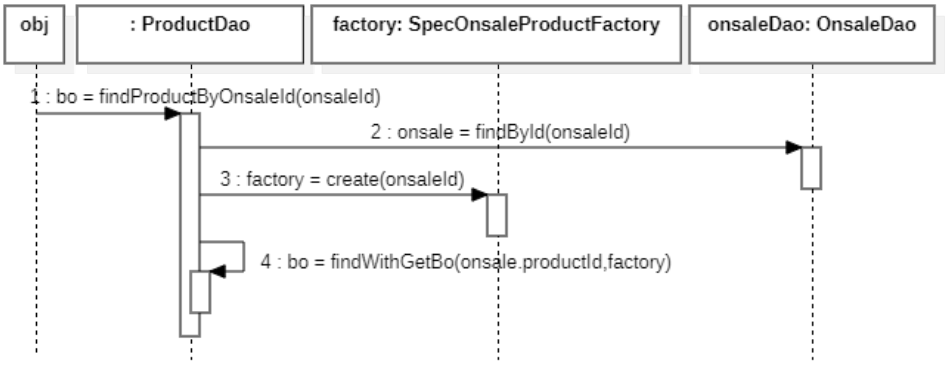

图 4：根据 onsaleId 获得 Product 的对象模型，关联指定的 Onsale 对象

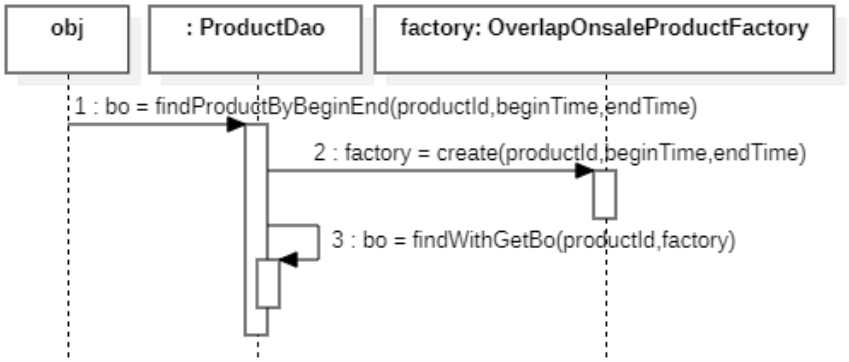

图 5：根据 productId，beginTime 和 endTime 获得 Product 的对象模型，关联在 beginTime 和 endTime 之间有效的 Onsale 对象

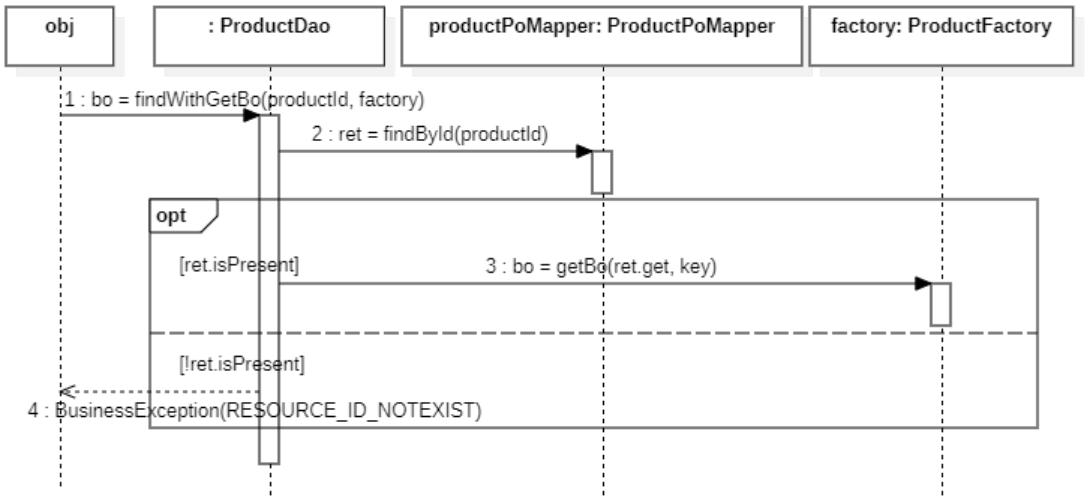

图 6：用不同的 factory 对象返回关联了不同 onsale 对象的 product 对象

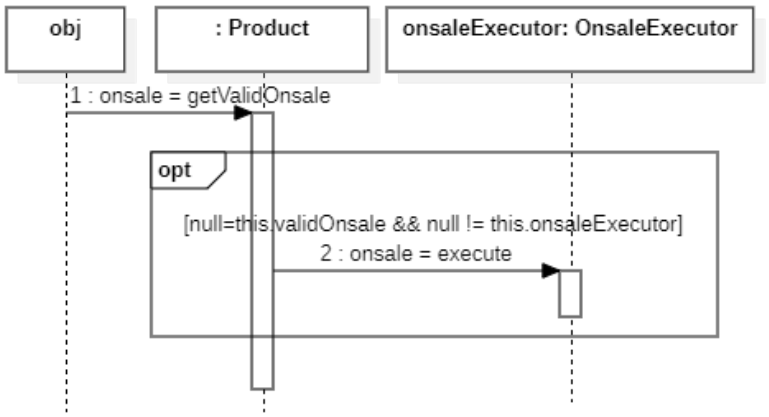

图 7：获得 product 对象关联的 onsale 对象

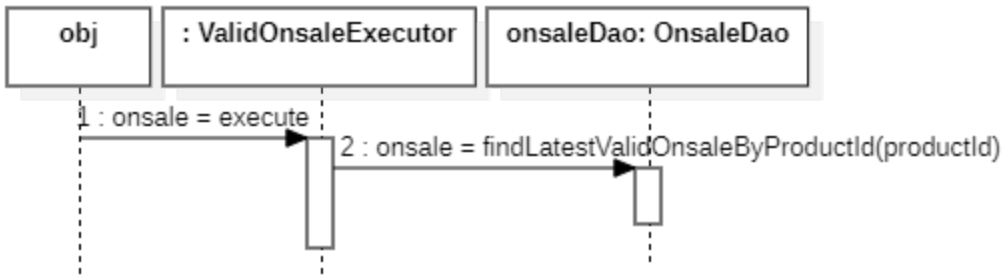

图 8：获得 product 最近有效的 onsale 对象

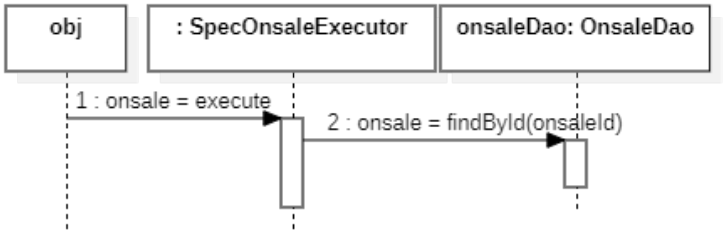

图 9：获得指定的 onsale 对象

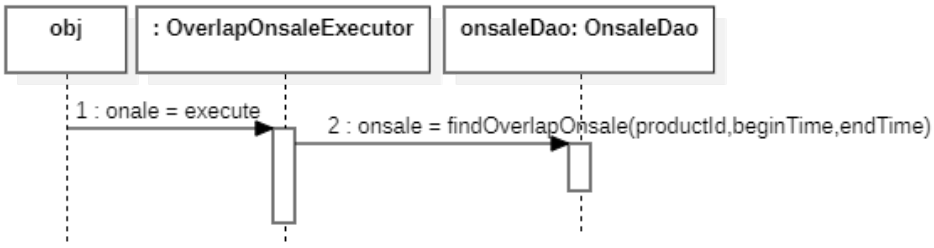

图 10：获得指定时间段内最早有效的 onsale 对象

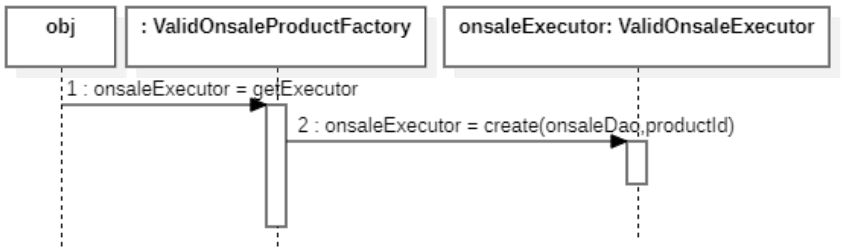

图 11：获得有效的 OnsaleExecutor 对象

图 12：获得指定的 OnsaleExecutor 对象

图 13：获得指定时间段的 OnsaleExecutor 对象

```
public abstract class ProductFactory implements GetBo<Product, ProductPo>{

    public Product getBo(ProductPo po,    Optional<String> redisKey){
        Product bo = Product.builder().id(po.getId()).creatorId(po.getCreatorId())
                        .creatorName(po.getCreatorName()).gmtCreate(po.getGmtCreate())
                        .gmtModified(po.getGmtModified()).modifierId(po.getModifierId())
                        .modifierName(po.getModifierName()).templateId(po.getTemplateId())
                        .shopId(po.getShopId()).shopLogisticId(po.getShopLogisticId())
                        .categoryId(po.getCategoryId()).goodsId(po.getGoodsId())
                        .name(po.getName()).skuSn(po.getSkuSn()).originPlace(po.getOriginPlace())
                        .status(po.getStatus()).unit(po.getUnit()).weight(po.getWeight())
                        .originalPrice(po.getOriginalPrice()).commissionRatio(po.getCommissionRatio())
                        .barcode(po.getBarcode()).build();
        this.setBo(bo);
        redisKey.ifPresent(key -> redisUtil.set(key, bo, timeout));
        return bo;
    }

    public void setBo(Product bo){
        bo.setCategoryDao(categoryDao);
        bo.setProductDao(ProductDao.this);
        bo.setShopDao(shopDao);
        bo.setOnsaleExecutor(this.getExecutor());
    }

    protected abstract OnsaleExecutor getExecutor();
}
```

图 14：ProductFactor 类代码