项目设计分享

第10组TRAP X00
陈俊达
2017年11月28日

目录

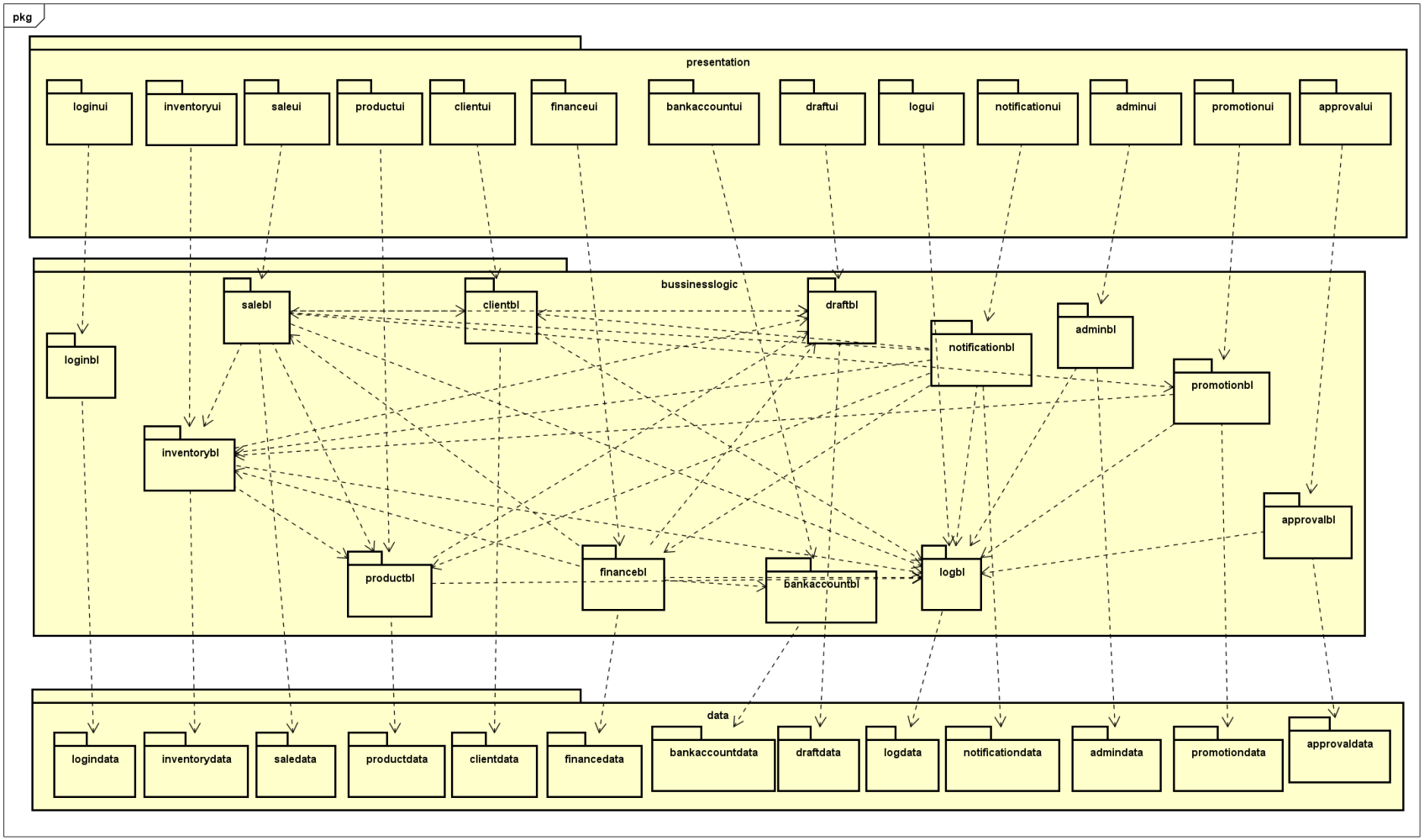
- 体系结构
- 详细设计

体系结构

体系结构

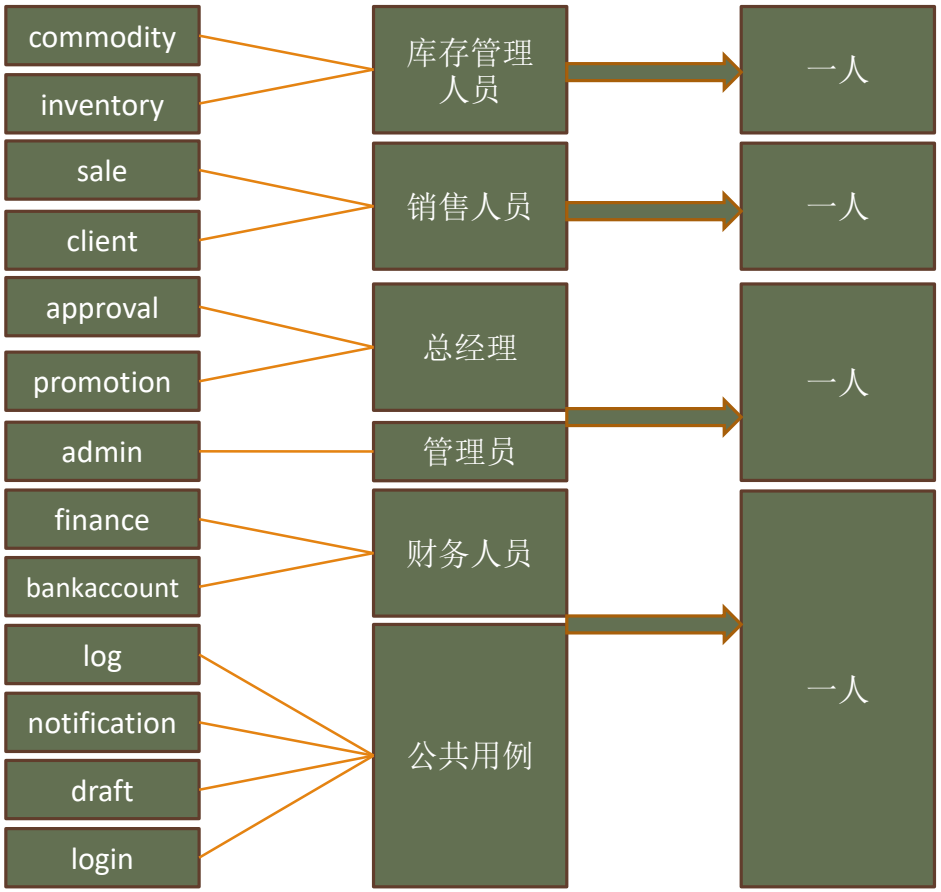
- 客户端-服务器端
- 分层
  - 客户端：展示层和逻辑层
  - 服务器端：数据层

逻辑包图


pkg
presentation
loginui
inventoryui
saleui
productui
clientui
financeui
bankaccountui
draftui
logui
notificationui
adminui
promotionui
approvalui
bussinesslogic
salebl
clientbl
draftbl
notificationbl
adminbl
promotionbl
loginbl
inventorybl
approvalbl
productbl
financebl
bankaccountbl
logbl
data
logindata
inventorydata
saledata
productdata
clientdata
financedata
bankaccountdata
draftdata
logdata
notificationdata
admindata
promotiondata
approvaldata

逻辑包特点

- Presentation包、bl包以及data包
- 一一对应（模块）
- 按角色分包，按角色分工
  - 现进度：data层已经结束



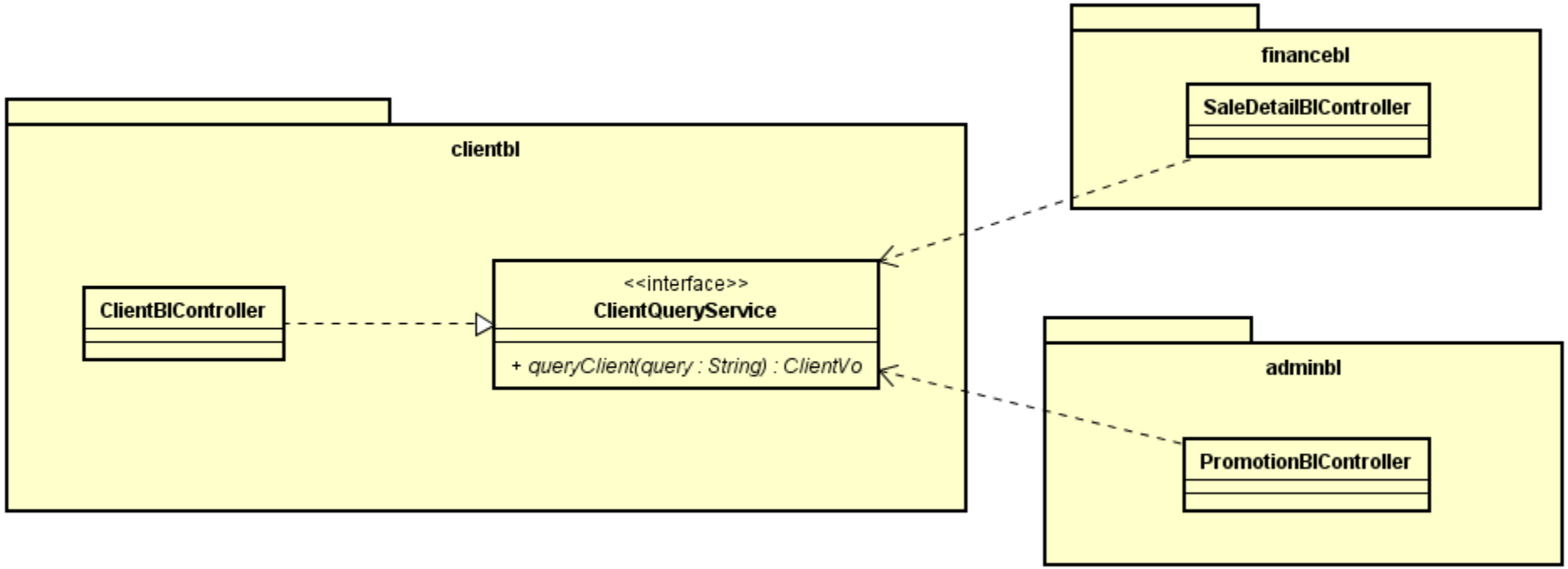
按角色分包

- 出发点：
  - 并行开发
  - 熟悉用例，节省时间
- 问题：
  - 接口、实现重复

接口、实现重复

- 解释
  - 角色间有重复的逻辑
  - **销售情况表**（财务人员）和**制定促销单据**（总经理）需要**客户信息**（进货销售人员）
- 解决
  - 层内包间接口调用

BL层获得用户信息


financebl
SaleDetailBlController
clientbl
<<interface>>
ClientQueryService
+ queryClient(query : String) : ClientVo
ClientBlController
adminbl
PromotionBlController

其他问题

- 权限控制？

总结

- 体系结构的特点：
  - Presentation包、bl包以及data包一一对应（模块）
  - 按角色分包，按角色分工
- 优势
  - 并行开发
  - 熟悉用例，节省时间

详细设计

详细设计特点

- Presentation层内业务逻辑
- 领域模型对象职责拆分
  - 全新的三层职责
- “3controller模式”

所有业务逻辑都要在BL层做？

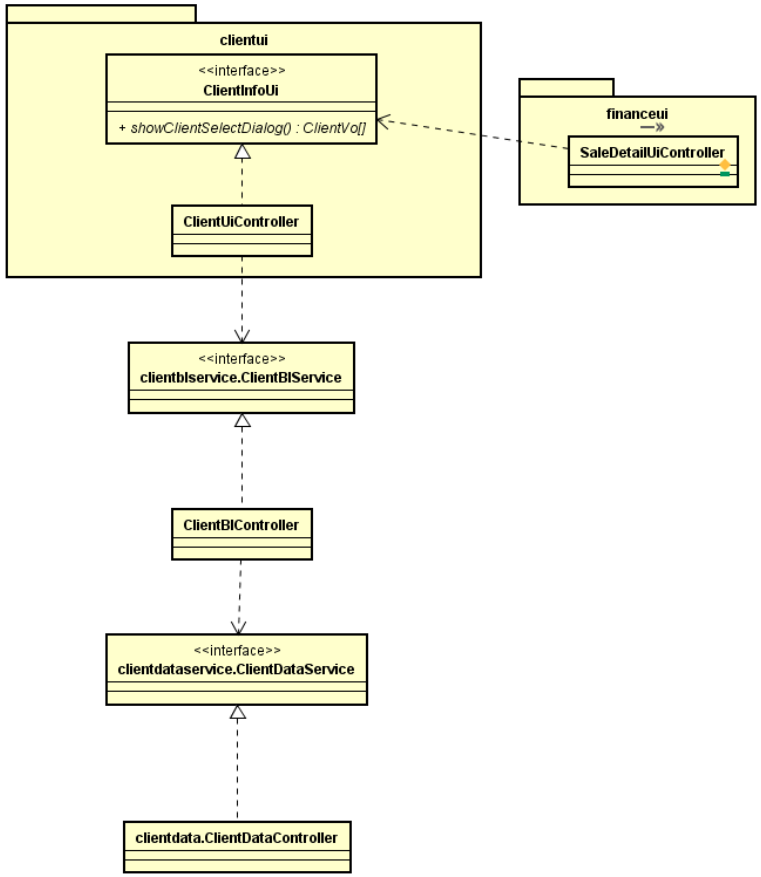
查看销售明细表（统计一段时间内商品的销售情况（应该就是查询销售出货单据记录），筛选条件有：**时间区间，商品名，客户，业务员，仓库。**显示符合上述条件的**商品销售记录**，以列表形式显示，列表中包含如下信息：**时间（精确到天），商品名，型号，数量，单价，总额。**需要支持导出数据。）

- 查看销售明细表需要筛选：
  - 客户信息
  - 商品信息
  - 职员信息（业务员）
  - 仓库信息

问题

- 太多层间接口！
- 职责不清
  - 选择客户的UI，为什么要财务人员做？
  - 需要财务人员熟悉客户相关定义
- 代码复用
  - 有很多用例都需要选择客户
  - 同样的逻辑写多次

展示层内交互



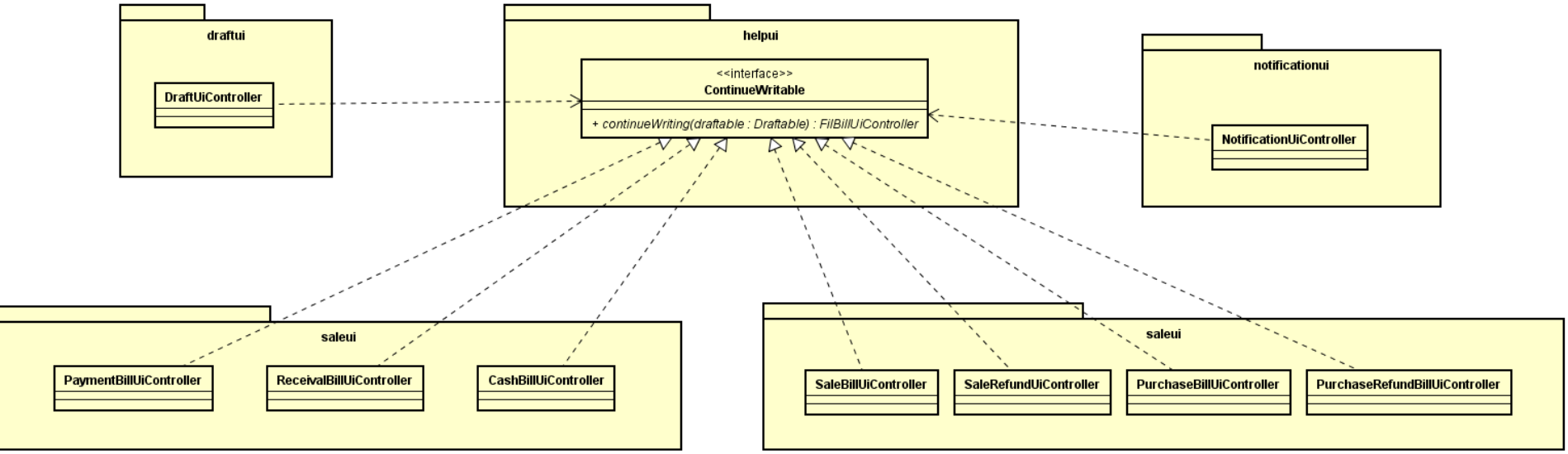
销售明细表用例筛选客户

```java
public interface SaleDetailBlService {
    /**
     * Queries SaleDetail.
     * @param query SaleDetail query conditions
     * @return SaleDetail that matches query conditions
     */

    SaleDetailVo query(SaleDetailQueryVo query);

    /**
     * Exports a SaleDetail.
     * @param detail SaleDetail to be exported
     * @return whether the operation is done successfully
     */

    ResultMessage export(SaleDetailVo detail);
}
```

销售明细表BlService

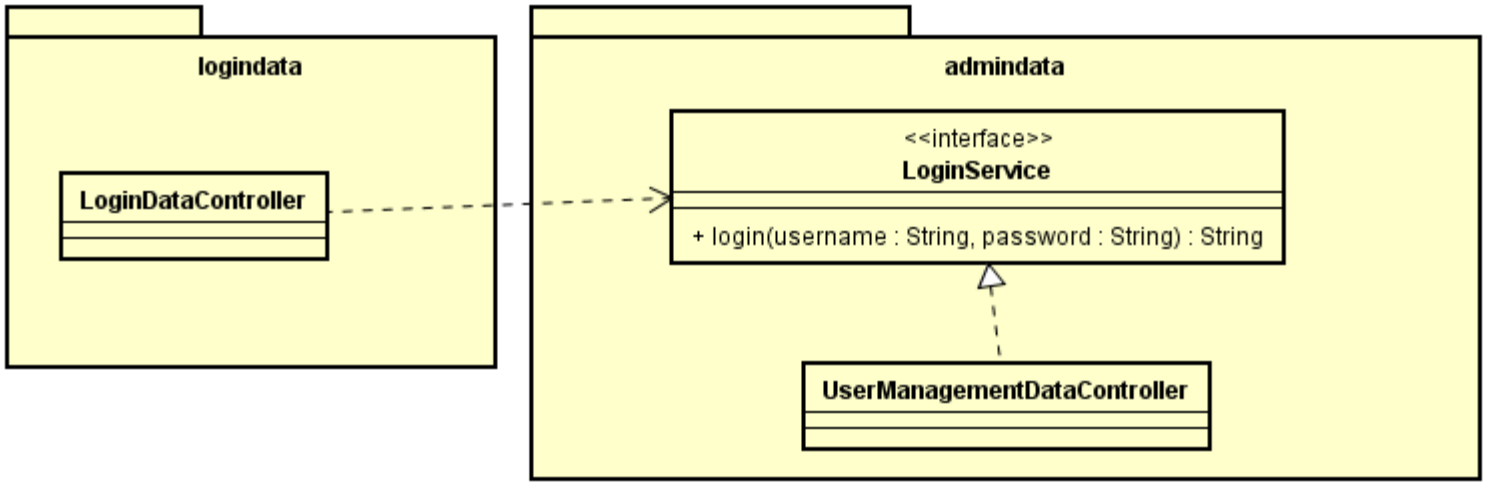
层内交互例子

- 继续填写草稿/修改审批结束但未入账的单据



层内交互例子

- （数据层）登录用到管理员用户账户管理数据



薄业务逻辑层

```java
public interface CashBillBlService {

    /**
     * Submits a CashBill.
     * @param bill CashBillVo to be submitted
     * @return whether the operation is done successfully
     */

    ResultMessage submit(CashBillVo bill);

    /**
     * Saves a half-completed CashBill as draft.
     * @param bill CashBillVo to be saved as a draft
     * @return whether the operation is done successfully
     */

    ResultMessage saveAsDraft(CashBillVo bill);

    /**
     * Gets the id for the next bill.
     * @return id for the next bill
     */
    String getId();

}
```

填写现金费用单模块的BlService

添加用户、添加现金款项等方法呢？

展示层

薄业务逻辑层

- 展示层包含部分业务逻辑
  - 维护当前正在填写的单据状态（Vo）
  - 向BL层提交单据、保存草稿
  - 草稿和通知跳往具体的单据填写界面
  - 查表时，表内显示简略信息，双击显示详细信息
  - ……

出发点

- “展示层是最容易变化的”
  - 只变化如何显示？
  - 业务逻辑同样也会变化
- 灵活，降低修改成本
- web前端

典型的领域模型对象

```
class CashBillItem {
    private String bankAccountId;
    private double quantity;
    private String content;
    public CashBillItem(String bankAccountId, double quantity, String content) {...}
}

class CashBillItemList {
    private ArrayList<CashBillItem> cashBillItems = new ArrayList<>();
    public void add(CashBillItem item) { cashBillItems.add(item); }
    public int count() { return cashBillItems.size(); }
}

class CashBill {
    private int id;
    private Date date;
    private CashBillItemList itemList = new CashBillItemList();

    public void addCashBillItem(CashBillItem saleItem) { itemList.add(saleItem); }
    public int count() { return itemList.count(); }
    public ArrayList<CashBillItem> getSaleItems() {...}
    public void submit() {}
}

class Caller {
    private static void main(String[] args) {
        CashBill cashBill = new CashBill();
        cashBill.addCashBillItem(new CashBillItem( bankAccountId: "123", quantity: 10, content: "haha"));
        System.out.println(String.format("There is/are %d item(s) currently in the list.", cashBill.count()));
        cashBill.submit();
    }
}
```

3. 提供服务

1. 维护自己的状态

2. 操作数据

3. 调用其他领域模型对象的服务

领域模型对象的职责

- 一个领域模型对象应该有如下职责：
  - 维护自己的状态（展示层通过BIService操作领域模型对象）
  - 操作数据（对Po的增删改查）
  - 调用其他领域模型对象的服务或向外提供服务
  - ……

疑问

- 查找现金费用单的职责的是谁的？
  - 对象？ =>如何生成第一个对象？
  - 类（静态方法）？ 其他类或对象？ =>多个类能操作数据库
- 展示层具有了逻辑后，逻辑对象的必要性

拆分领域模型对象的职责

- 维护状态：Vo
- 操作数据、提供服务、调用服务：Controller

问题：Vo到服务的对应

- 等待审批的单据内容存在其对应的单据数据库里，由对应单据模块管理
- 存在两种通知（单据审批通过和其他通知）
- 单据审批通知Vo包含一个单据BillVo（所有单据的父类）
- 问：如何通过BillVo对象直接进入对应数据库，将数据库中此单据条目的状态改为被丢弃？
- 逻辑对象可以直接操作数据库。

解决

- 单据丢弃服务（NotificationAbandonService）

```java
public interface NotificationAbandonService {
    /**
     * Abandons a bill.
     * @param id id for the bill
     * @return whether the operation is done successfully
     */
    ResultMessage abandon(String id);
}
```

```java
public class CashBillBlController implements CashBillBlService, NotificationActivateService, NotificationAbandonService, DraftDe
```

解决

- BillVo提供获得对应服务的方法，子类实现。

```
/**
 * Gets the NotificationAbandonService corresponding to this type of bill. Overrides to meet the specific bill type.
 * @return NotificationAbandonService
 */
public abstract NotificationAbandonService notificationAbandonService();
```

```
@Override
public NotificationAbandonService notificationAbandonService() {
    return CashBillBlFactory.getNotificationAbandonService();
}
```

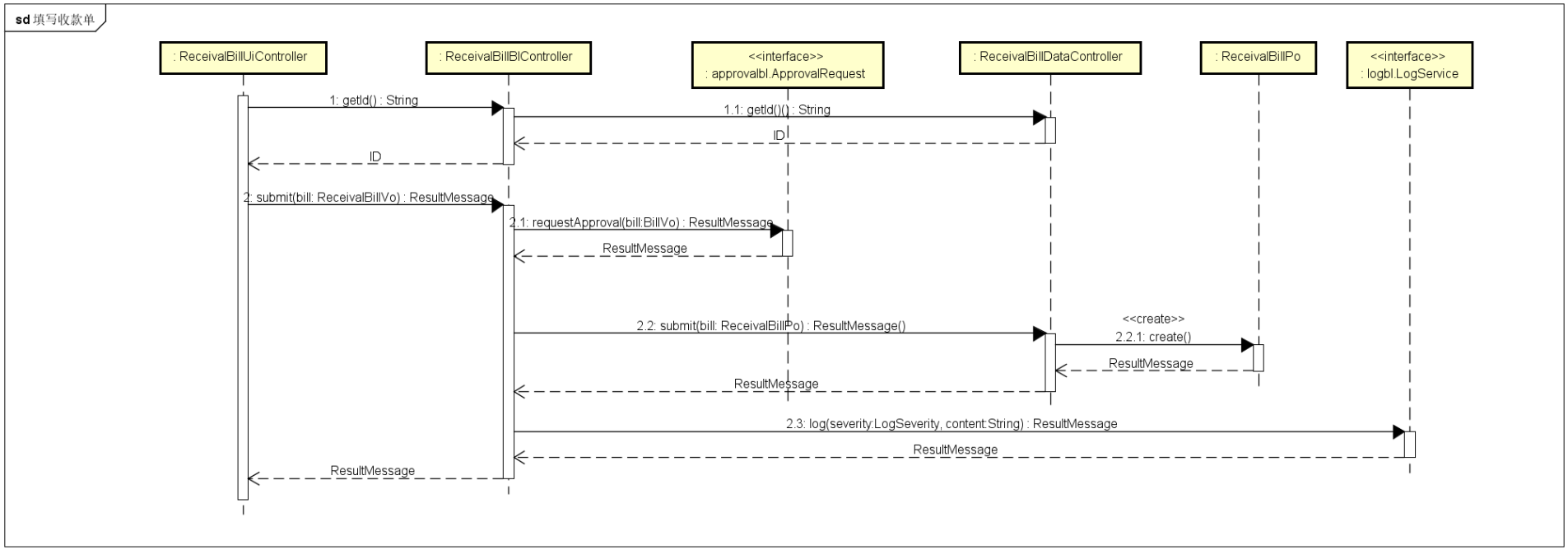
解决

- 通知BlController直接调用服务

```
/**
 * Abandons a bill.
 *
 * @param notification notification with a bill
 * @return whether the operation is done successfully
 */
@Override
public ResultMessage abandon(BillApprovalNotificationVo notification) {
    NotificationAbandonService service = notification.getBill().notificationAbandonService();
    service.abandon(notification.getBill().getId());
    return ResultMessage.Success;
}
```

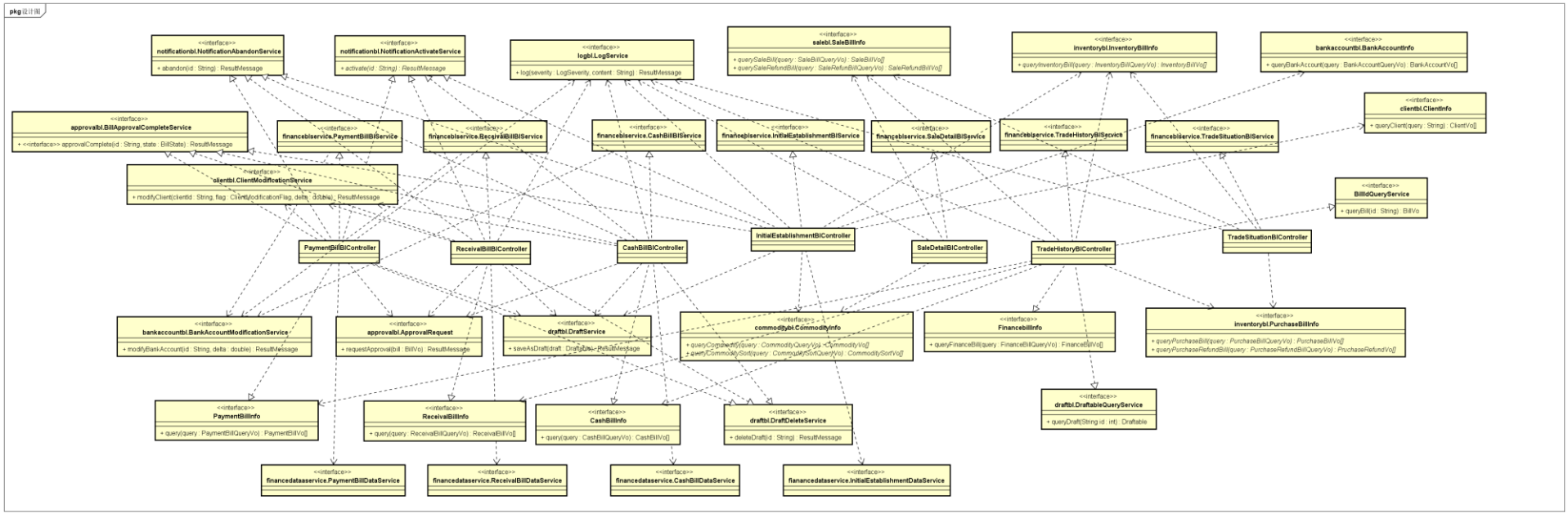
BL层新职责

- 通过对象交互实现功能
- 对象交互通过接口



缺点

- 混乱的BL层



违反单一职责（？）

新的职责

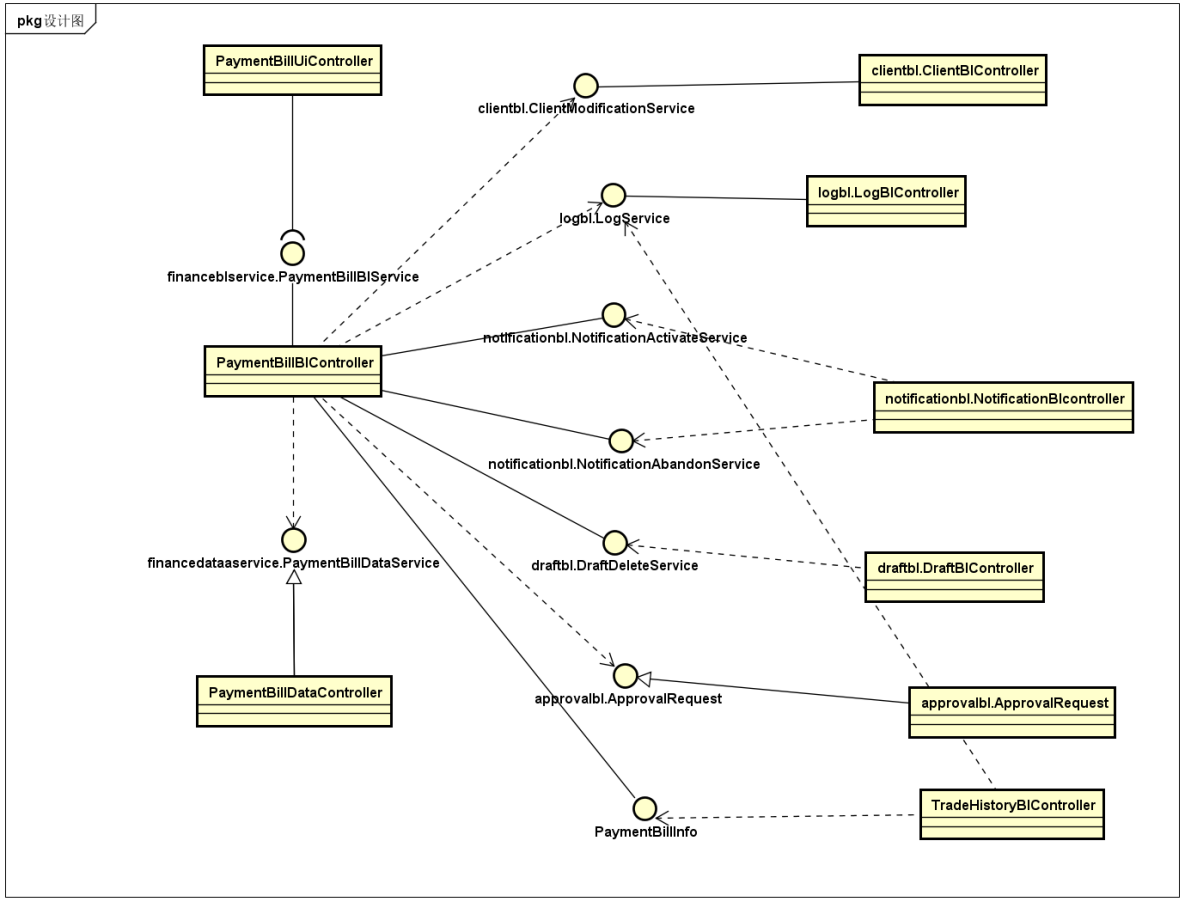
- 类/对象新职责：
  - UiController：实现界面功能（而不仅是展示）
  - BlController：通过与其他对象、服务交互实现功能
  - DataController：操作数据
  - Vo：维护状态、传递信息

设计特点

- Presentation层内业务逻辑
- 领域模型对象职责拆分
- “3controller模式”

3controller模式

- 每层一个单例Controller，实现此包所有接口，依赖其他包接口



优势

- 层次关系明显
  - 只有一个对象能够操作下一层
- 模块职责清晰
  - 每个模块实现所有自己的相关方法
  - 不需要了解其他模块的实现、定义等
  - 通过接口与其他模块交互

“注册表”模式

- 问题：
  - 客户信息、单据、促销策略可作为草稿内容（Draftable）
  - 所有草稿Po保存草稿内容的ID，放在一张表里，以枚举类型作为区分
  - 草稿Po转到Vo时，不同类型需要通过不同的接口查找具体的草稿对象

```
@DatabaseTable(tableName = "Draft")
public class DraftPo {
    @DatabaseField(generatedId = true)
    private int id;
    @DatabaseField
    private Date saveTime;
    @DatabaseField
    private String saverId;
    @DatabaseField
    private String draftableId;
    @DatabaseField
    private DraftType draftType;

    public DraftPo(Date saveTime, String saverId, String draftableId, DraftType draftType) {
        this.saveTime = saveTime;
        this.saverId = saverId;
        this.draftableId = draftableId;
        this.draftType = draftType;
    }
```

“注册表”模式

- 草稿查找服务（DraftableQueryService）、实现、注册

```java
public interface DraftableQueryService {
    /**
     * Queries draft with id.
     * @param id id
     * @return draft
     */
    Draftable queryDraft(String id);
}
```

```java
public class TradeHistoryBlController implements TradeHistoryBlService, FinanceBillInfo,  DraftableQueryService {
```

```java
public abstract class BillVo implements Draftable {
    static {
        DraftableQueryServiceRegistry.register(DraftType.Bill, BillDraftQueryServiceFactory.getQueryService());
    }
```

“注册表”模式

- 注册表

```
public class DraftableQueryServiceRegistry {
    private static HashMap<DraftType, DraftableQueryService> registry = new HashMap<>();

    /**
     * Registers DraftType with service.
     * @param draftType DraftType
     * @param service DraftableQueryService
     */
    public static void register(DraftType draftType, DraftableQueryService service) {
        registry.put(draftType, service);
    }

    /**
     * Queries Draftable with registered service.
     * @param draftType DraftType
     * @param id id for the draft
     * @return Draftable
     */

    public static Draftable queryDraftable(DraftType draftType, String id) {
        return registry.get(draftType).queryDraft(id);
    }

}
```

“注册表”模式

- 去掉switch
- 用于：
  - 通知
    - 两种通知（单据审批结束和其他）
    - 单据审批结束通知Vo需要查询单据详细信息
  - Data层处理用户增删改查
    - 5种用户5张表
    - 根据传入Po实际类型使用不同的Dao对不同的数据库进行操作

总结

- 详细设计的特点：
  - Presentation层内业务逻辑
  - 领域模型对象职责拆分
  - “3controller模式”
- 优势：
  - 展示层更加灵活
  - 代码复用
  - 层次关系明显
  - 职责清晰
    - 类/对象
    - 模块

谢谢！