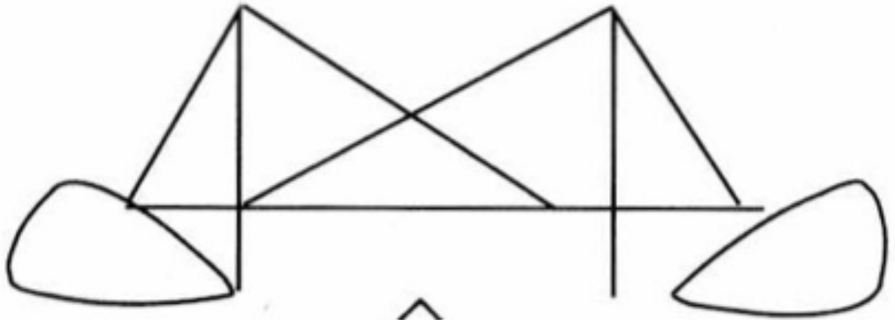
详细设计的总结与补充

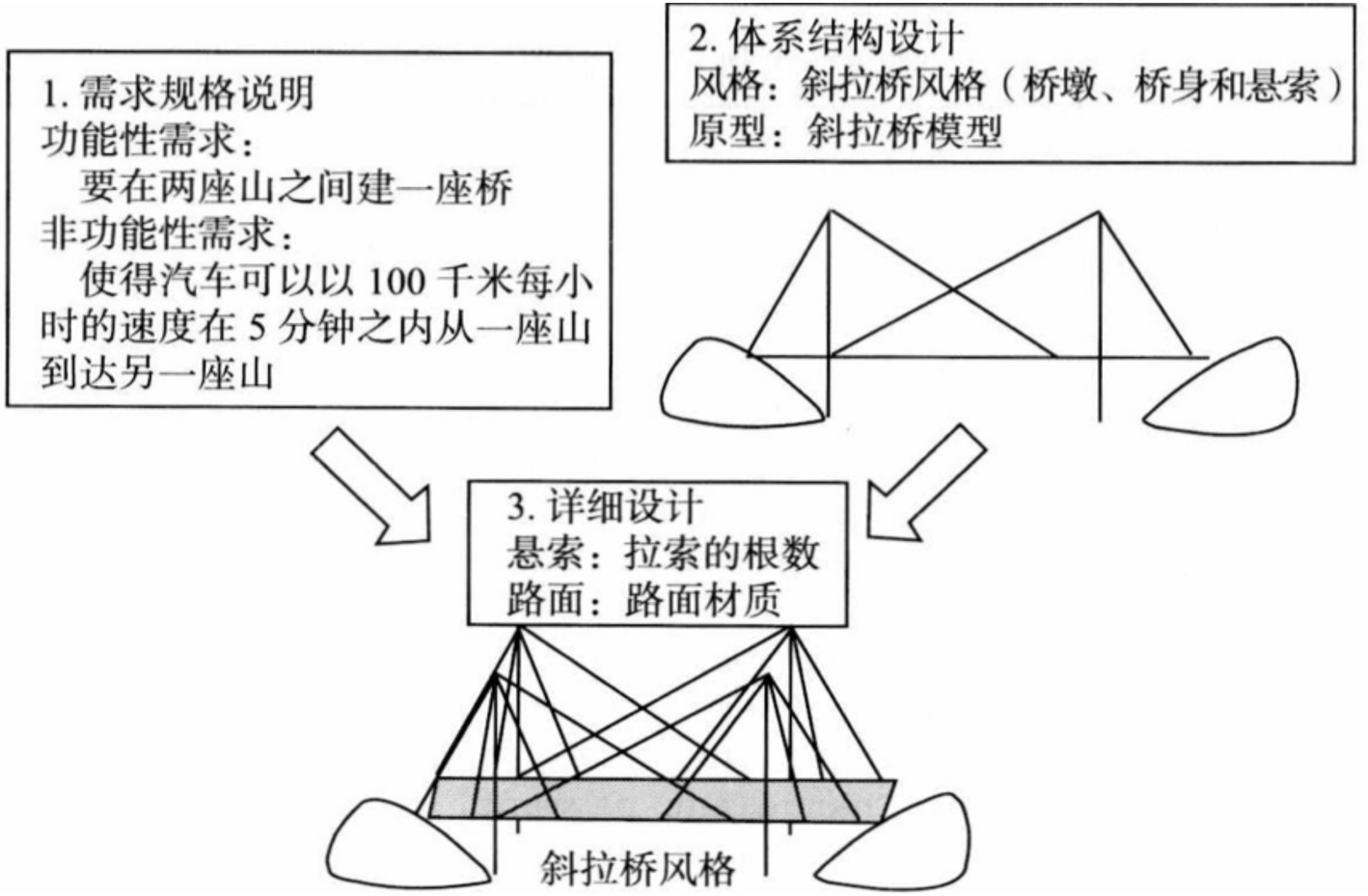
1. 详细设计的输入和输出

1. 需求规格说明
功能性需求：
　要在两座山之间建一座桥
非功能性需求：
　使得汽车可以以 100 千米每小时的速度在 5 分钟之内从一座山到达另一座山

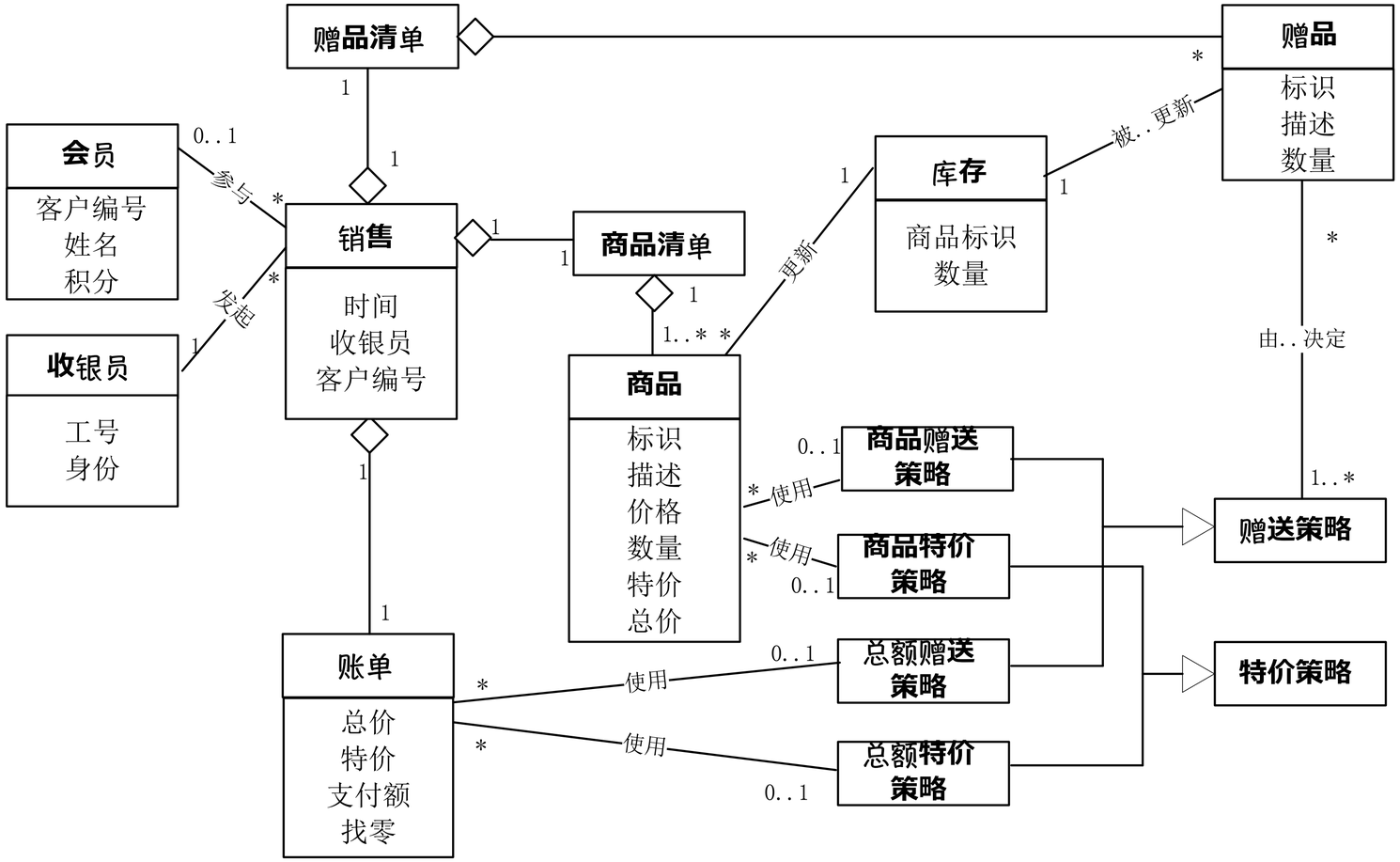
2. 体系结构设计
风格：斜拉桥风格（桥墩、桥身和悬索）
原型：斜拉桥模型



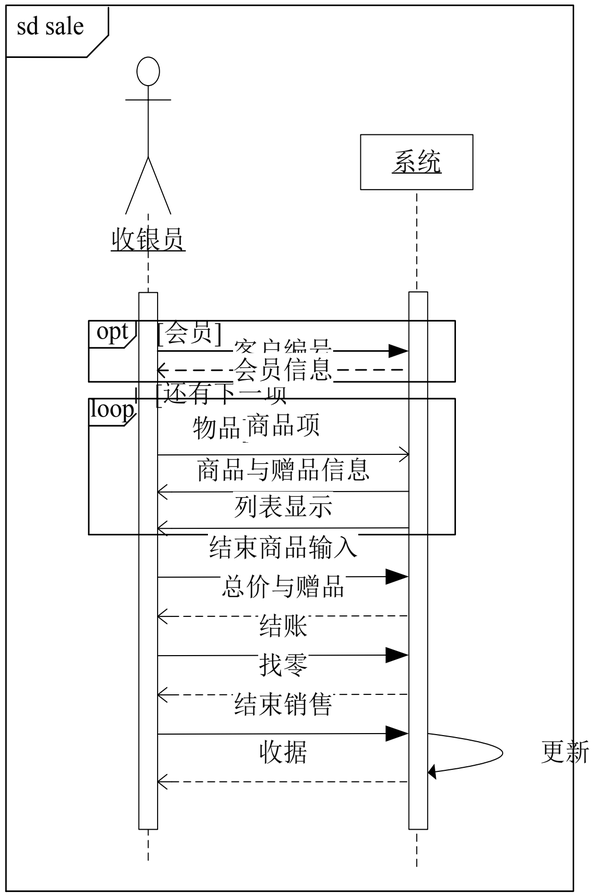
详细设计的输入



从需求、体系结构设计到详细设计


赠品清单
赠品
标识
描述
数量
会员
客户编号
姓名
积分
销售
时间
收银员
客户编号
商品清单
库存
商品标识
数量
收银员
工号
身份
商品
标识
描述
价格
数量
特价
总价
商品赠送策略
商品特价策略
总额赠送策略
总额特价策略
赠送策略
特价策略
账单
总价
特价
支付额
找零
参与
关心
更新
被..更新
由..决定
使用
使用
使用
使用
0..1
1..*
*
1

需求：分析类图

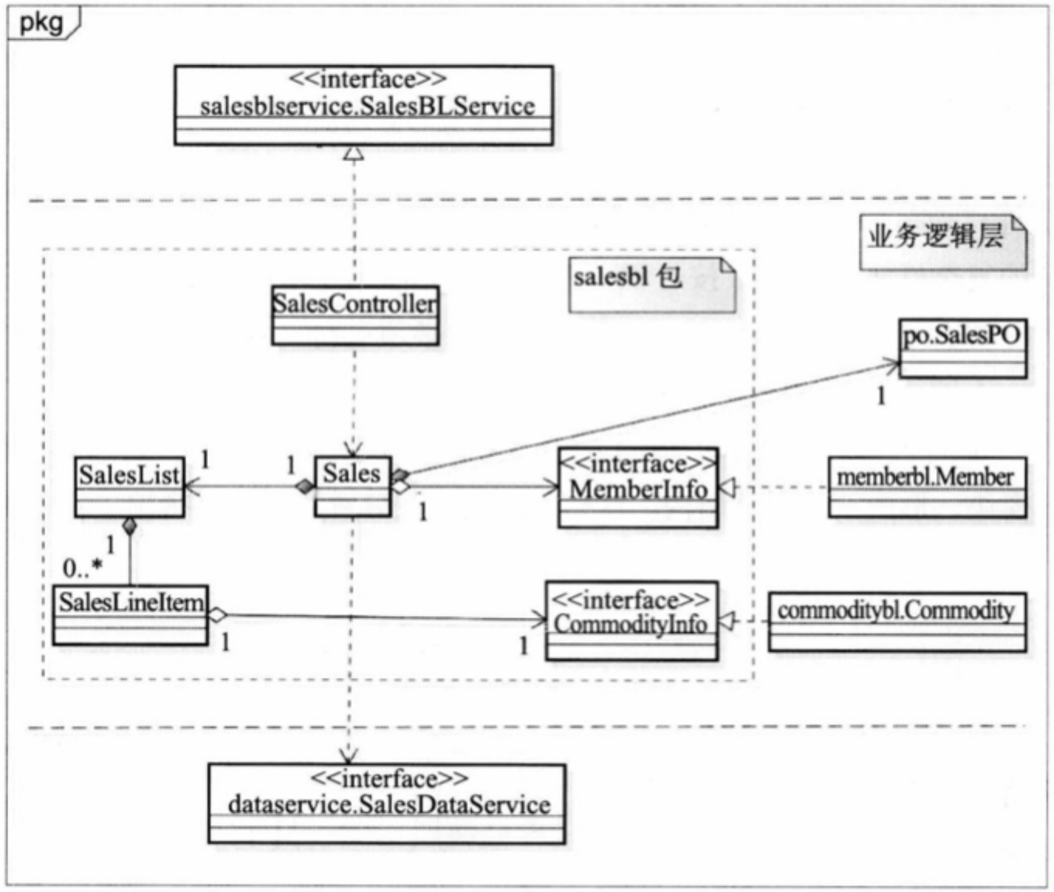

sd sale
系统
收银员
opt
[会员]
会员信息
loop
[还有下一项]
物品-商品项
商品与赠品信息
列表显示
结束商品输入
总价与赠品
结账
找零
结束销售
收据
更新

需求：系统顺序图

```
// 被 Presentation 层调用的接口
public interface SalesBLService {
    public CommodityVO getCommodityByID(int id);
    public MemberVO getMember();
    public  CommodityPromotionVO  getCommodityPromotionByID(int
        commodityID);
    public boolean addMember(int id);
    public boolean addCommodity(int id, int quantity);
    public int getTotal(int mode);
    public int getChange(int payment);
    public void endSales();

    // 调用 DataService 层的接口
    public interface SalesDataService extends Remote {
        public void init()throws RemoteException;
        public void finish()throws RemoteException;
        public void insert(SalesPO po)throws RemoteException;
        public void delete(SalesPO po)throws RemoteException;
        public void update(SalesPO po)throws RemoteException;
        public SalesPO find(int id)throws RemoteException;
         public  ArrayList<PO>  finds(String  field,  int  value)throws
RemoteException;
}
```

软件体系结构：构件之间的接口


pkg
<<interface>>
salesblservice.SalesBLService
业务逻辑层
salesbl 包
SalesController
po.SalesPO
1
SalesList
1
1
Sales
<<interface>>
MemberInfo
memberbl.Member
1
1
0..*
SalesLineItem
1
<<interface>>
CommodityInfo
commoditybl.Commodity
1
<<interface>>
dataservice.SalesDataService

详细设计的输出

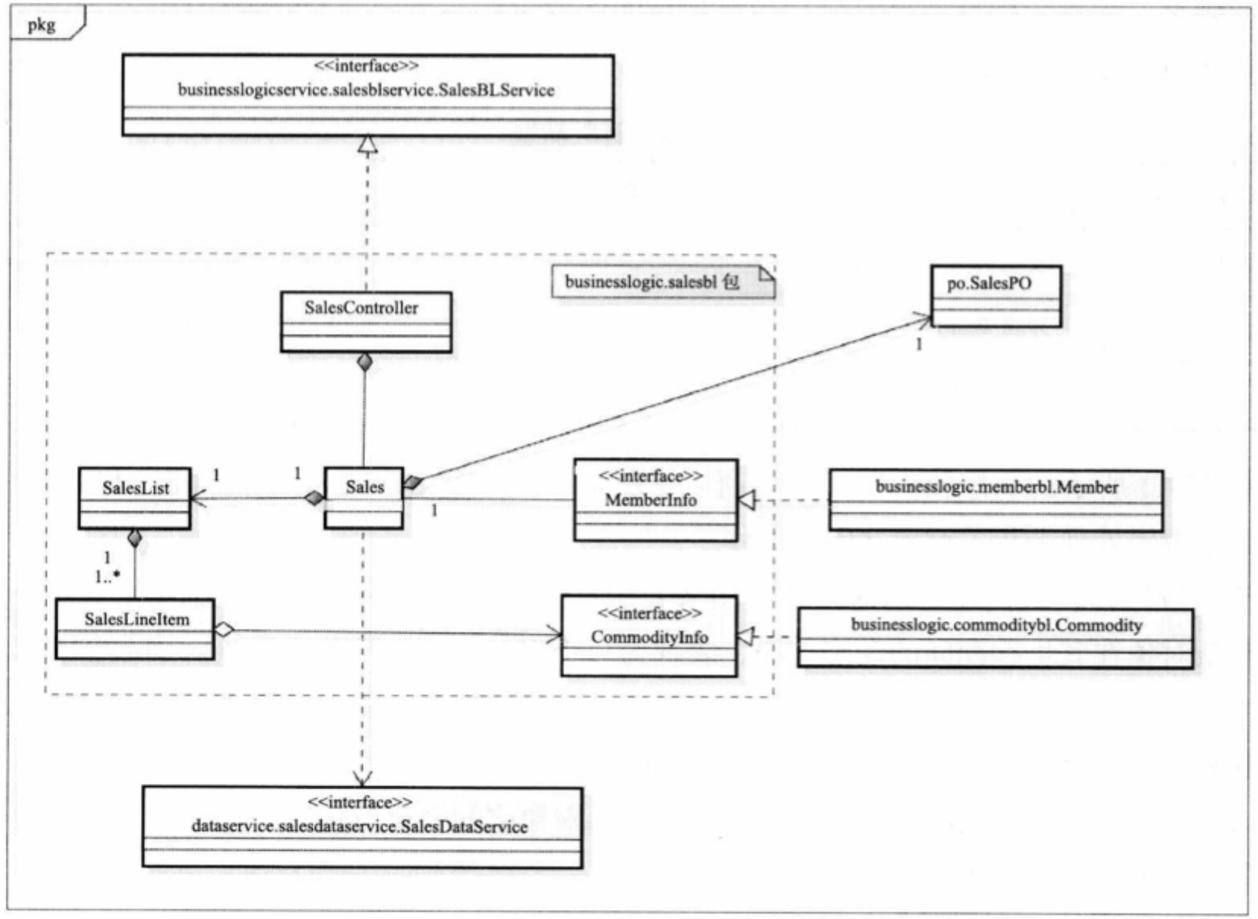
2. 详细设计的过程

面向对象设计的过程

- 设计模型建立
  - 通过职责建立静态设计模型
    - 抽象类的职责
    - 抽象类之间的关系
    - 添加辅助类
  - 通过协作建立动态设计模型
    - 抽象对象之间协作
    - 明确对象的创建
    - 选择合适的控制风格
- 设计模型重构
  - 根据模块化的思想进行重构，目标为高内聚、低耦合
  - 根据信息隐藏的思想进行重构，目标为隐藏职责与变更
  - 利用设计模式重构

辅助类

- 接口类
- 记录类(数据类)
- 启动类
- 控制器类
- 实现数据类型的类
- 容器类



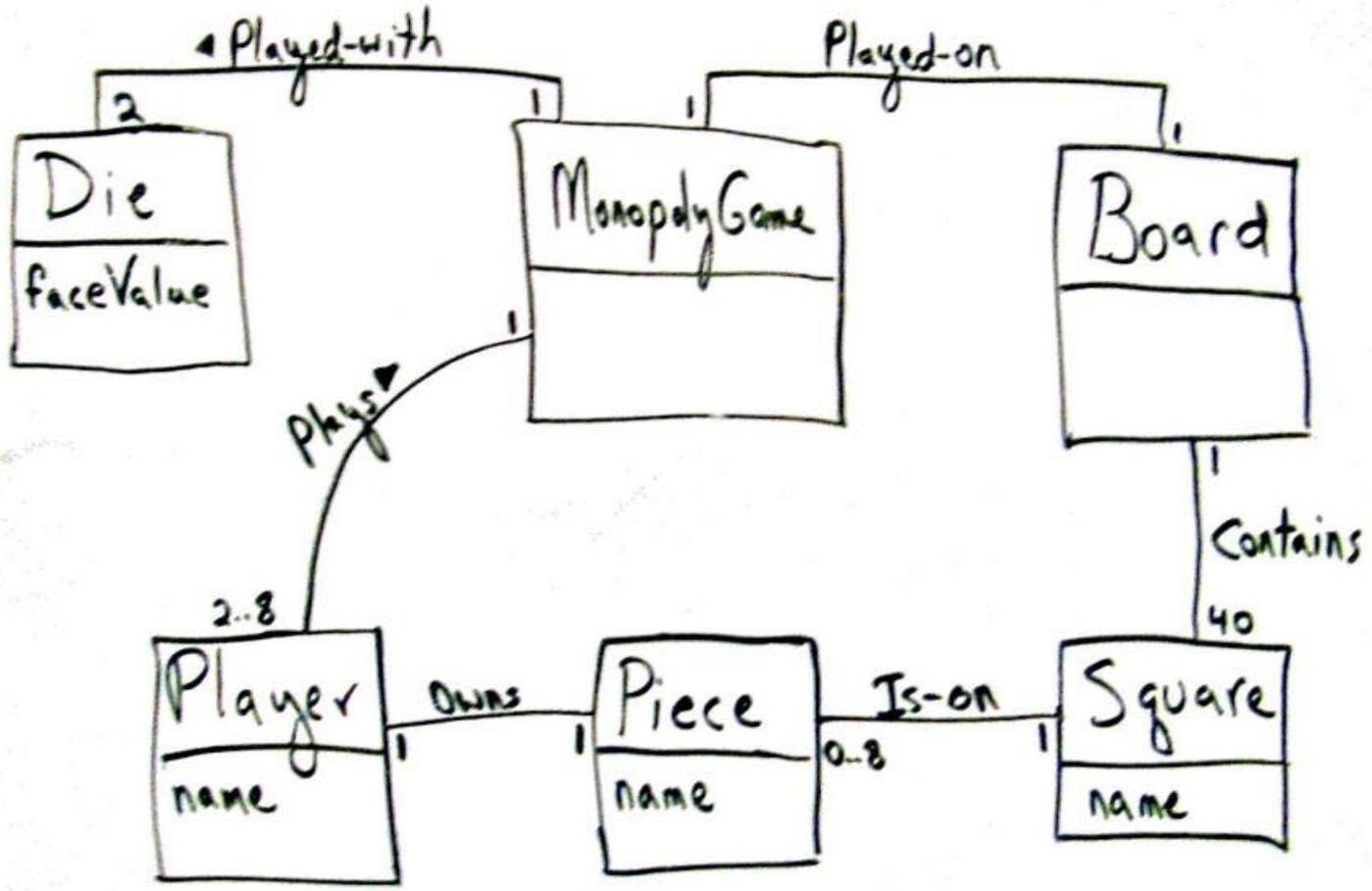
添加辅助类后的设计模型

3. GRASP Pattern

UML和模式应用

GRASP Patterns

- General Responsibility Assignment Software Patterns
- Not ‘design patterns’, rather fundamental principles of object design
- Focus on one of the most important aspects of object design: assigning responsibilities to classes

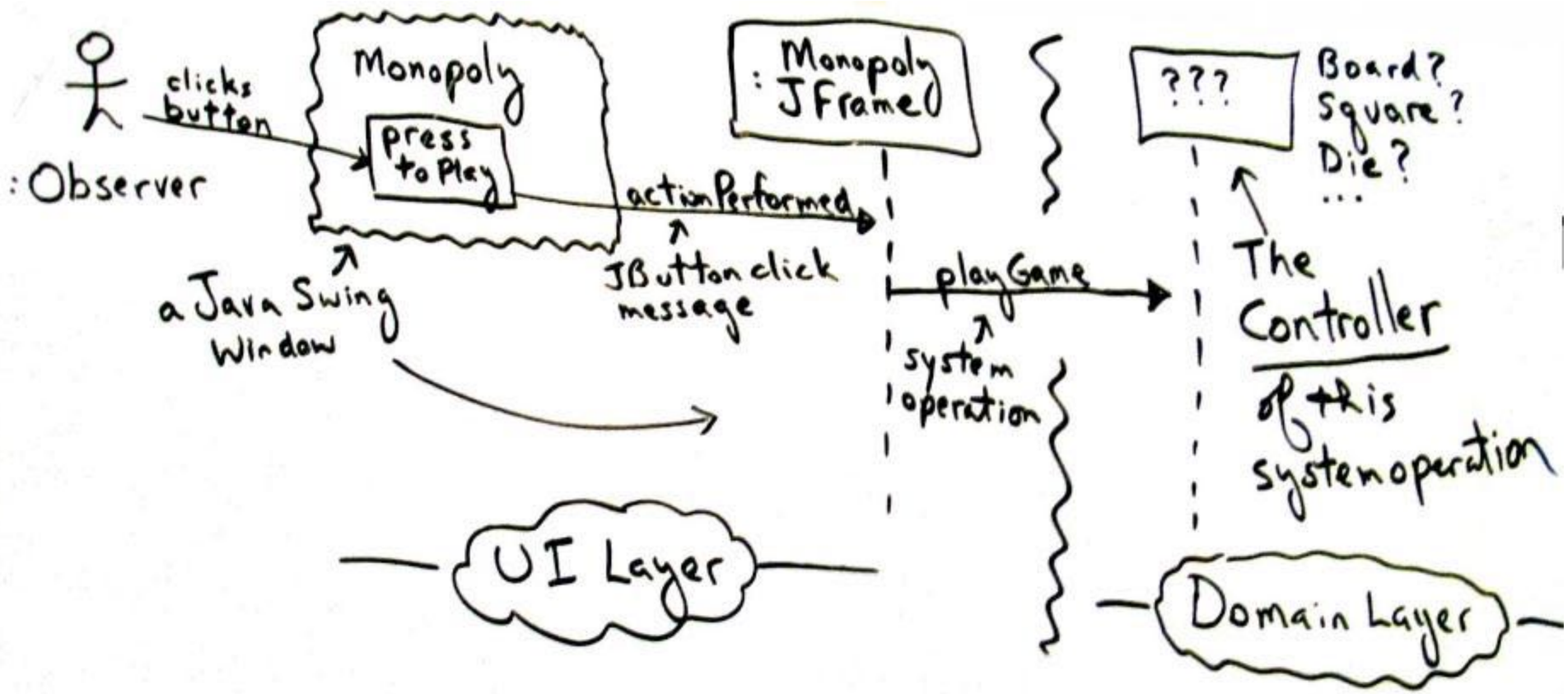


Who creates the Squares/Piece/Player?

Piece的创建

- Player?
- Board?

- 两者的区别是什么?

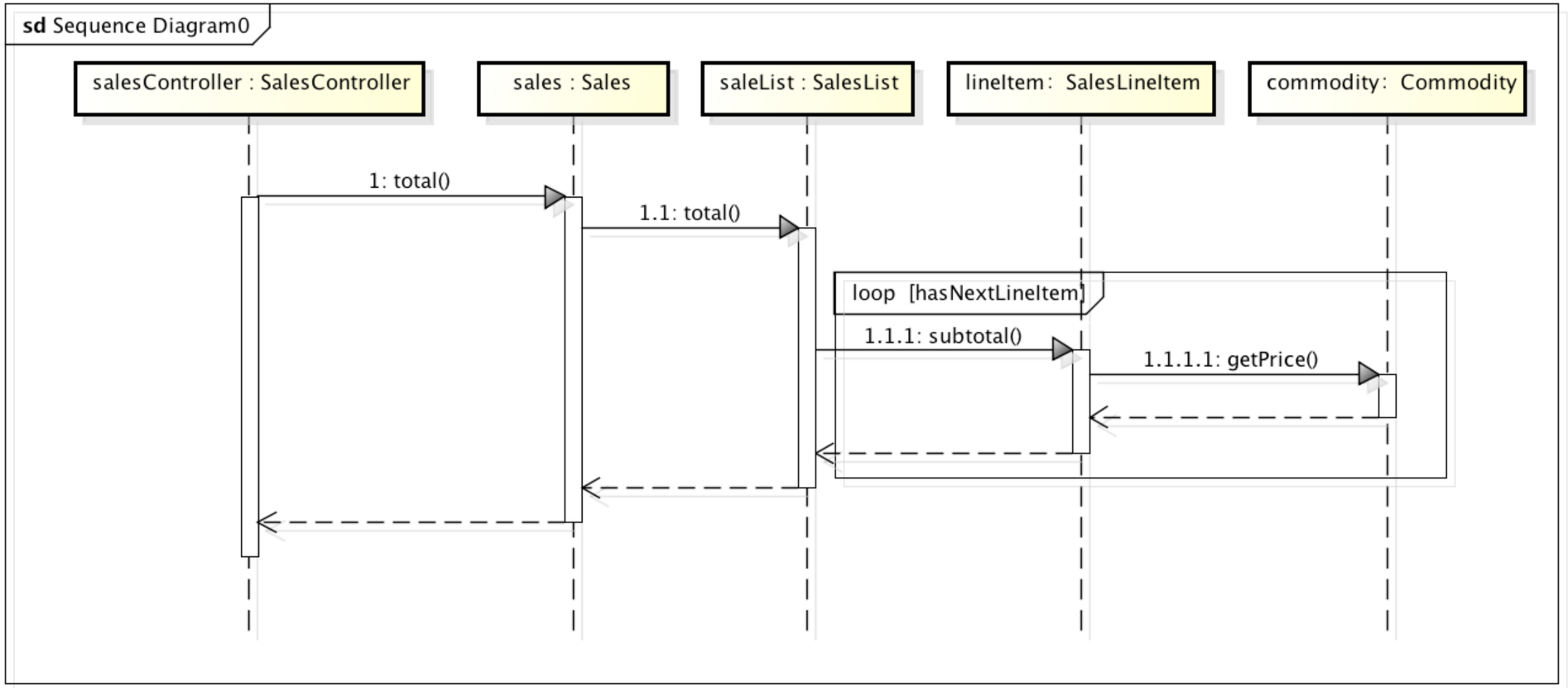


Who is the controller of playGame operation?

4. 详细设计的集成测试

详细设计的集成测试

- 类间协作的集成测试
  - 重点针对复杂逻辑（交互比较多）
  - 自顶向下或者自底向上的集成
- Mock Object
  - 不是stub
- 测试用例


sd Sequence Diagram0
salesController : SalesController
sales : Sales
saleList : SalesList
lineItem : SalesLineItem
commodity : Commodity
1: total()
1.1: total()
loop [hasNextLineItem]
1.1.1: subtotal()
1.1.1.1: getPrice()
powered by astah*

类间协作的集成测试

```
public class MockCommodity extends Commodity{
        double price;
        public MockCommodity ( double p){
                price=p;
        }
        public getPrice ( ){
                return price;
        }
}
```

MockObject

类似于Stub，但是更简单

```
public class TotalIntegratoin nTester {
        @Test
        public void testTotal () {
            MockCommdity commodity1 = new MockCommdity (50);
            MockCommdity commodity2 = new MockCommdity (40);
            SalesLineItem salesLineItem1 =
                                    new SalesLineItem (commodity1, 2);
            SalesLineItem salesLineItem2
                                = new SalesLineItem (commodity2, 3);

            Sales sale=new Sales();
            sale.addSalesLineItem(salesLineItem1) ;
            sale.addSalesLineItem(salesLineItem2) ;

            assertEquals (220, sale.total () );
         }
}
```

集成测试代码