BUPT
TSEG

软件工程 模型与方法
Models & Methods of SE

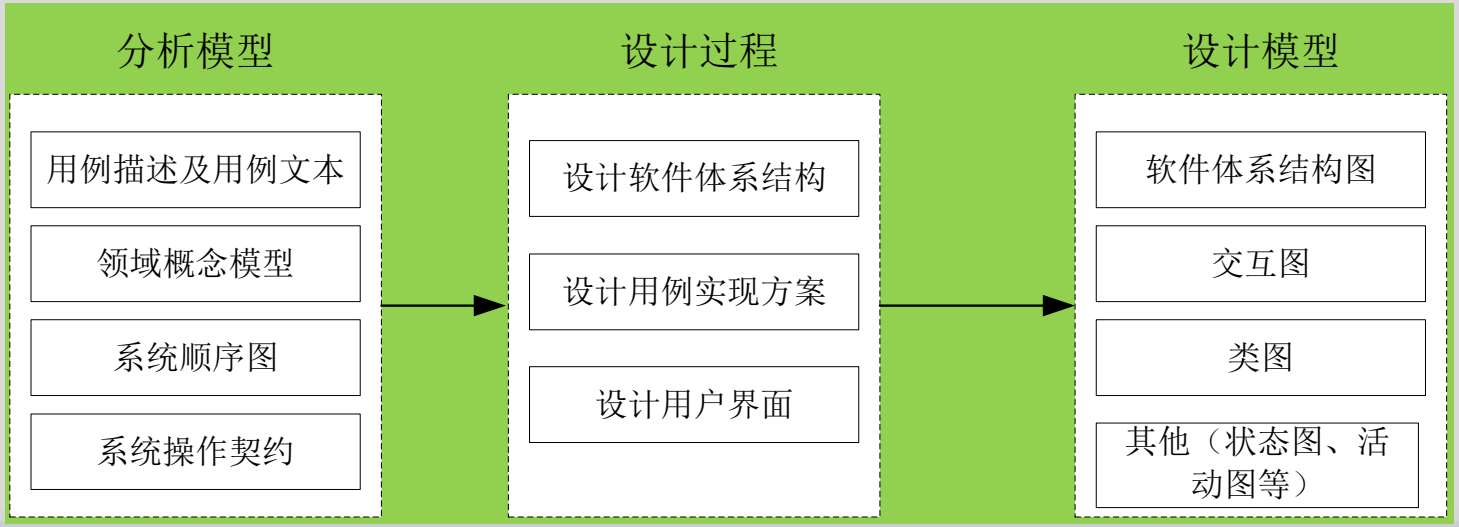
面向对象设计方法

本章内容

- 面向对象设计综述
- 模型的层次化
- 面向对象设计原则
- 设计用例实现方案

面向对象设计综述

- 面向对象的设计：以需求分析阶段的用例模型和领域模型为基础，运用UML构建软件系统结构，通过一系列设计模型说明用例的实现过程。
- 主要设计活动：
  - 选择合适的软件架构；
  - 根据架构通过UML 交互图描述每个用例的实现过程；
  - 最终给出以UML类图表示的能够满足所有用例的系统静态结构；
  - 根据系统的设计原则进行优化。



面向对象的设计与UML

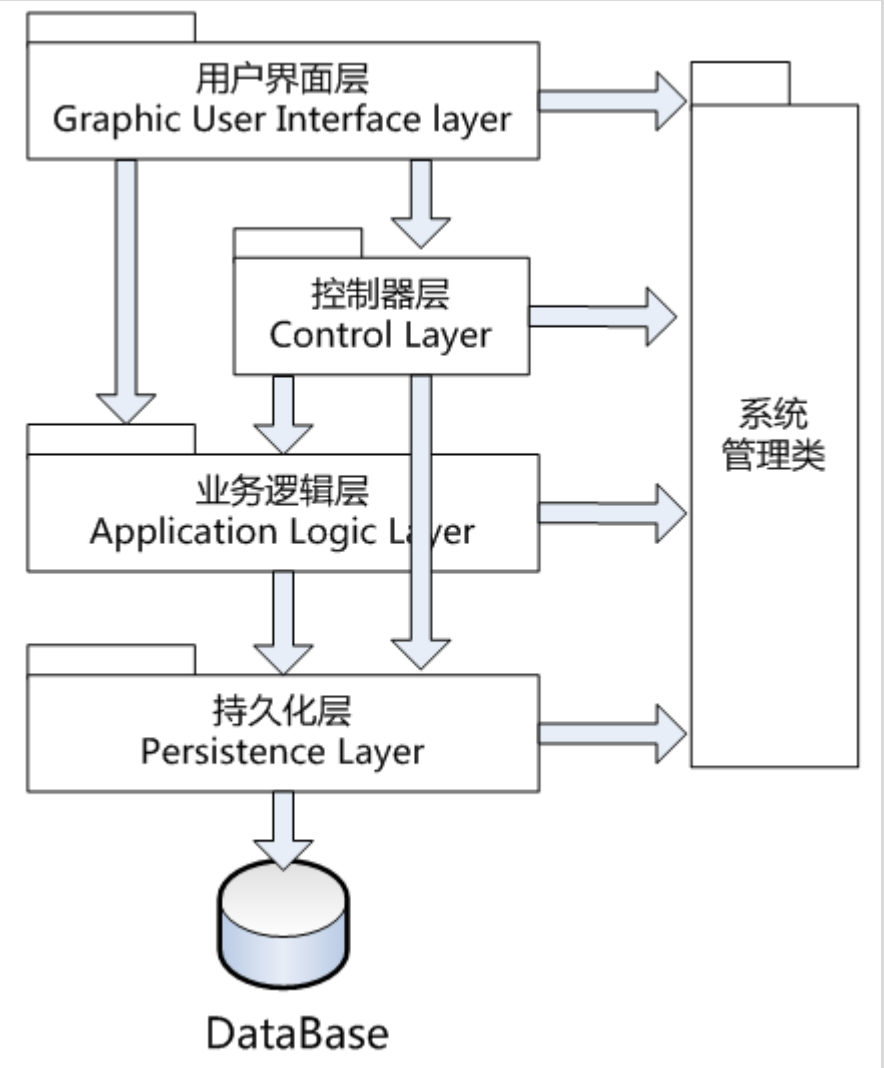
- 软件概要设计步骤
  - 选择合适的软件架构；
  - 系统的动态结构设计：
    - 用例实现过程设计，针对用例对应的SSD中的每个系统事件，运用UML的 sequence diagram / collaboration diagram 给出符合该系统事件定义的操作契约的内容；
    - 如果软件对象具有多种不同的职责（主要考虑对应于不同的用例）的情况下，需要运用 state machines diagram 对该软件对象进行状态迁移的设计；
  - 系统的静态结构设计
    - 对所有用例或者子系统级别的用例的交互图进行归纳，运用UML的 Class diagram 给出系统的静态结构；
- 软件详细设计
  - 针对系统静态结构中每个对象的方法，运用UML activity diagram 对其进行逻辑结构的设计

面向对象设计的关键步骤

- 在确定软件框架结构的基础上，进行以下内容的设计
  - 发现对象（发现软件类）：根据需求和选择的架构和模式确定系统由哪些对象构成；
  - 确定对象属性：明确该对象应该具有的特征属性；
  - 确定对象行为：明确对象应具有的功能和职责；
  - 确定对象之间的关系：根据系统顺序图及操作契约以及选择的架构和模式明确系统是如何相互协作完成功能需求的交互过程；

（基于BS结构）模型的层次化

- 层次化的设计模型是面向对象方法基于软件体系结构风格的一种方案选择。层次化的设计模型符合面向对象的设计原则，并使系统易于扩展和维护。
  - 用户界面层：（用例）系统功能的各种界面表现形式。
  - 控制器层：用于协调、控制其他类共同完成用例规定的功能或行为。
  - 业务/应用层：实现用例要求的各种系统级功能；
  - 持久化层：用于保存需要持久化存储的数据对象；
  - 系统层：为应用提供操作系统相关的功能，通过把特定于操作系统的特性包装起来，使软件与操作系统分离，增加应用的可移植性。



用户界面层

- 用户界面层指与用户进行交互的部分，包含应用程序中用户界面部分的代码。
- 系统与用户的界面可以以多种形式出现：
  - 图形用户界面GUI
  - 命令行界面
  - 其他交互界面（语音等）
- 尽量将用户界面层与系统的业务逻辑分离，专门处理系统与用户的交互。
- 用户与系统的交互方式发生变化，系统的基本业务逻辑不需改变；系统业务逻辑变化，在交互内容不变的情况下，用户界面不需要进行改变。

控制器层对象设计原则

- 问题来源：SSD中的系统事件应由哪个或哪些软件对象负责接收和处理？（界面层的请求发给哪个或哪些软件对象？）
- 参考答案：
  - 方案一：请求直接发给对应的业务对象进行逻辑处理；
  - 方案二：请求被某一个对象接收并转发给其他业务对象进行逻辑处理；
- 方案二的对象称为控制器类，不同用例的系统事件可以对应不同的控制器类。这一类对象对应的层次称为控制器层。

业务/应用层对象设计原则

- 问题来源：用例对应的核心功能应由哪些软件对象具体实现?
- 参考答案：尽量参考领域模型中的概念类对软件对象进行定义和命名。
- 问题来源：这些软件对象应位于分层结构中的哪一层?
- 参考答案：这些软件对象代表具体的业务功能需求，应位于业务逻辑层。
- 在面向对象分析阶段，已经识别出了问题域中重要的概念，该阶段关注的是概念的本质含义以及属性。
- 在面向对象设计阶段，将会对这些概念增加操作，并进行必要的修改和调整，使之成为设计模型中业务/领域层中的类。
- 这也是为什么说OOA和OOD采用一致的表示法，OOA和OOD之间不存在结构化方法中分析与设计的鸿沟，两者能够紧密衔接。

持久化层对象设计原则

- 问题来源：需要持久保存的业务数据存放在哪里？
- 参考答案：该软件对象的职责就是管理（增删改查）经过业务逻辑对象处理后的需要持久保存的数据，又能与业务逻辑的功能相分离保持其独立性，又能与数据库保持同步，这些对象称为持久化对象。
- 对象持久化：将对象状态永久保存到物理存储介质中。

为何引入持久化类

- 引入持久层的目的在于当数据存储机制或策略发生变化的时候，能减少维护工作。
- 目前大部分系统都是采用数据库作为存储介质。但数据库肯定会改变，包括：
  - 数据库升级
  - 从一种数据库移动到另一种数据库
  - 数据模式变化，如增加字段、修改字段名称、改变字段类型等
- 持久层将对数据库的操作类封装起来，提供专门数据管理功能，向业务/领域对象提供持久化服务，从而使数据库变化对业务领域的影响的范围局部化。
- 无论持久存储策略如何变化，业务/领域类都不会受影响，从而增加了应用程序的可维护性、可扩展性和可移植性。

系统层

- 系统层提供对操作系统和非面向对象资源的访问。
- 系统类将操作系统提供的系统调用封装起来，生成系统访问类，例如Java语言中的文件流类库。上层业务逻辑直接访问系统类。而不直接访问系统调用。
- 这样当程序需要在不同操作系统平台上进行移植时，只需要修改少数系统类就可以。

设计用例实现方案

- 面向对象的设计过程也可称为用例实现的设计
- 用例实现：在设计模型中描述分层结构中相互协作的软件对象如何实现用例的各个特定场景，包括所有的成功和失败场景。
- 用例实现的设计方案：根据选择的分层结构，结合需求分析的结果找到各层次对应的软件对象，并给出这些对象的交互场景以实现用例的要求。
  - 使用UML 的sequence/collaboration diagram 进行绘制。
- 进一步根据找到的软件对象在交互图中的接收和发送的消息，确定软件对象应该具有软件方法和属性。
- 完成从分析模型到设计模型的的转变过程。

类职责分配(Grasp)模式

- 设计类的来源有两部分。
  - 核心逻辑由领域模型中的概念类转换而来
  - 另一部分则是为实现而新增的一些类，如负责对象持久化的类、负责通信的类。
- 每一个设计类都有明确的职责，分为两种类型：
  - 了解型（knowing）职责（自己干自己的事）细分为三类：对象要了解自己私有的封装数据；了解相关联的对象；了解能够派生或者计算的事物。
  - 行为型（doing）职责（自己干自己能干的事）。细分为三类：对象自身要能执行一些行为，如创建一个对象或者进行计算；对象要能启动其他对象中的动作；对象要能控制或协调其他对象中的活动。
  - 职责的内聚（自己只干自己的事）：目的是提高内聚降低耦合，减少不必要的关联关系

面向对象的设计模式

- 对象的职责通过调用对象的方法来实现。将职责分配给一个对象还是多个对象，是分配给一个方法还是多个方法要受到职责粒度的影响。
- 面向对象设计最关键的活动是正确地给对象分配职责。
- 模式是面向对象软件的设计经验，是可重用的设计思想，它描述了在特定环境中反复出现的一类设计问题，并提供经过实践检验的解决这类问题的通用模式。
- 模式定义了一组相互协作的类，包括类的职责和类之间的交互方式。

模式的组成

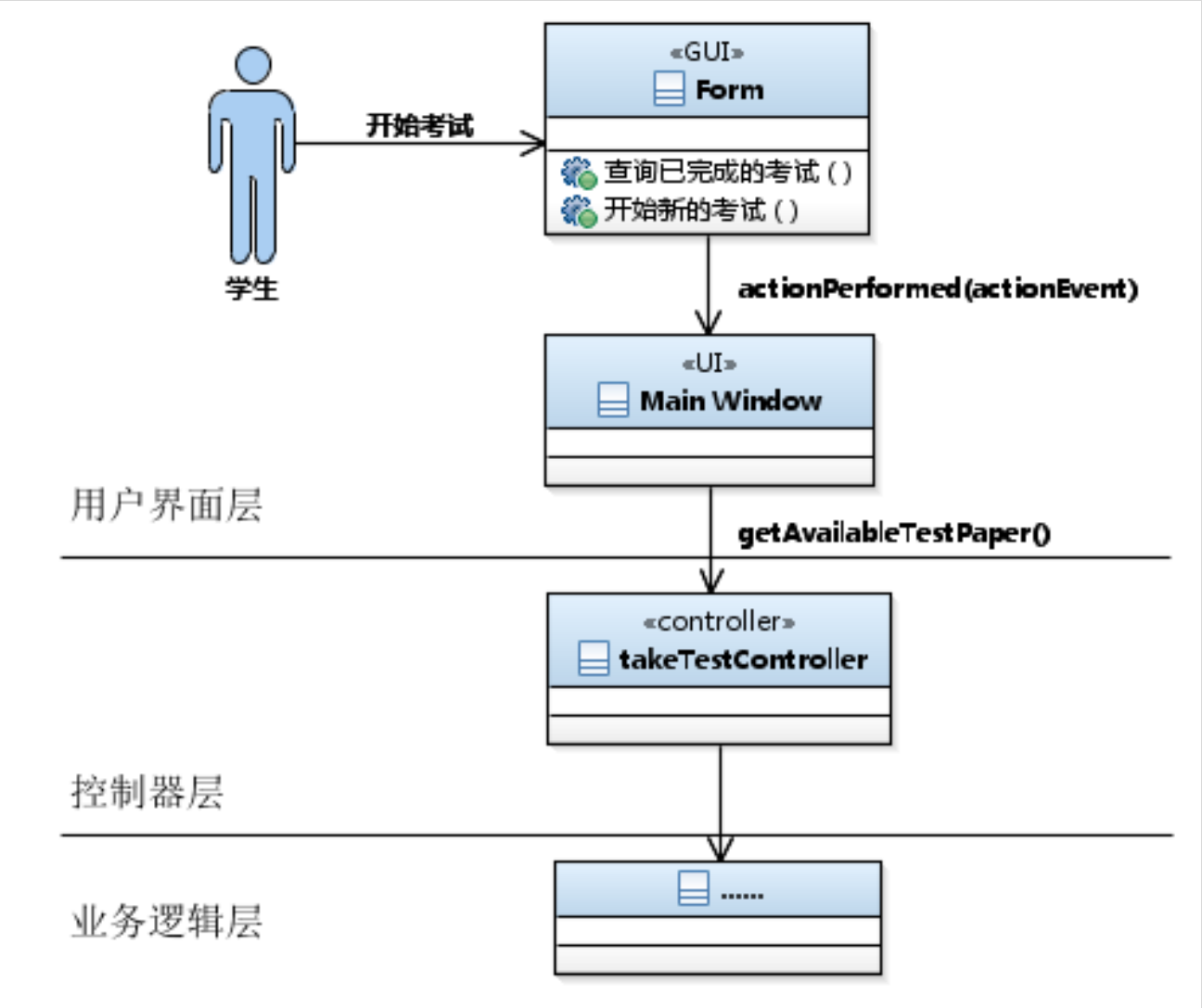
- 模式名称：一个助记名，用一两个词描述模式的问题、解决方案和效果；
- 问题：描述了何时使用模式。或者说模式的使用问题域；
- 解决方案：描述了设计的组成部分、组成部分之间的相互关系及各自的职责和协作方式；
- 效果：描述了模式应用的效果和使用模式应权衡的问题。

控制器(Controller)模式

- 问题来源：第一个接收系统事件的软件对象是什么？哪个软件对象负责接收和处理一个系统输入事件?
- 解决方案：把接收和处理系统事件的职责分配给位于控制器层的对象
  - 它代表整个系统（系统简单且不复杂），称为外观（facade）控制器;
  - 它代表一个发生系统事件的用例场景，这个类通常命名为“<用例名>控制器”，称为用例控制器或者会话控制器。
  - 在相同的用例场景中使用同一个控制器类处理所有的系统事件;
  - 一次会话是与一个参与者进行交谈的一个实例。

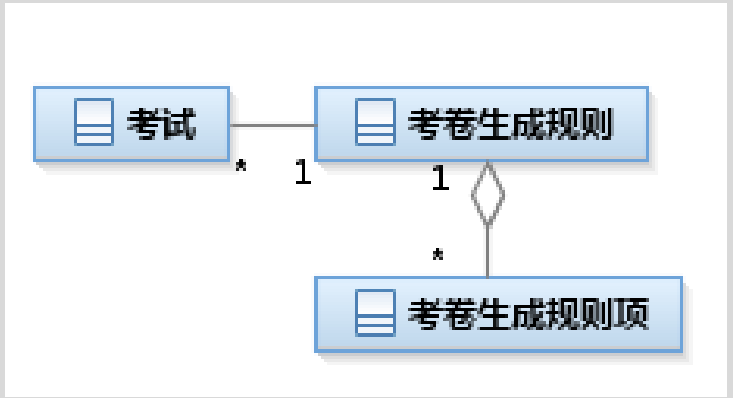
使用控制器的指导原则

- 当一个系统不具有“太多”的系统事件，或者用户接口不可能将事件消息重定向到其他控制器时，选择外观控制器是合适的。这时，外观控制器相当于一个应用的封面，隔离了用户接口和应用逻辑。
- 如果外观控制器由于职责过多而变得“臃肿”的时候，应该选择用例控制器。
  - 如果选择了用例控制器，那么每一个用例都有一个不同的控制类，而且只有一个，以便维护用例的状态。用例控制器可以实现有一定执行顺序的系统操作。
- 不论是外观控制器还是用例控制器，它们只是接收系统事件消息，并没有实现系统操作的职责，系统操作应该委托给领域对象处理。



创建者(Creator)模式

- 问题来源：哪个对象应该负责产生类的实例？（操作契约中对象实例的创建）
- 如果符合下面的一个或者多个条件，则可将创建类A实例的职责分配给类B(B创建A)。
  - B聚合（aggregate）或包含（contain）对象A；
  - B记录（record）对象A；
  - B密切使用对象A；
  - B拥有创建对象A所需要的初始化数据（B是创建对象A的信息专家）。
- 创建者模式体现了低耦合的设计思想，是对迪米特法则的具体运用。



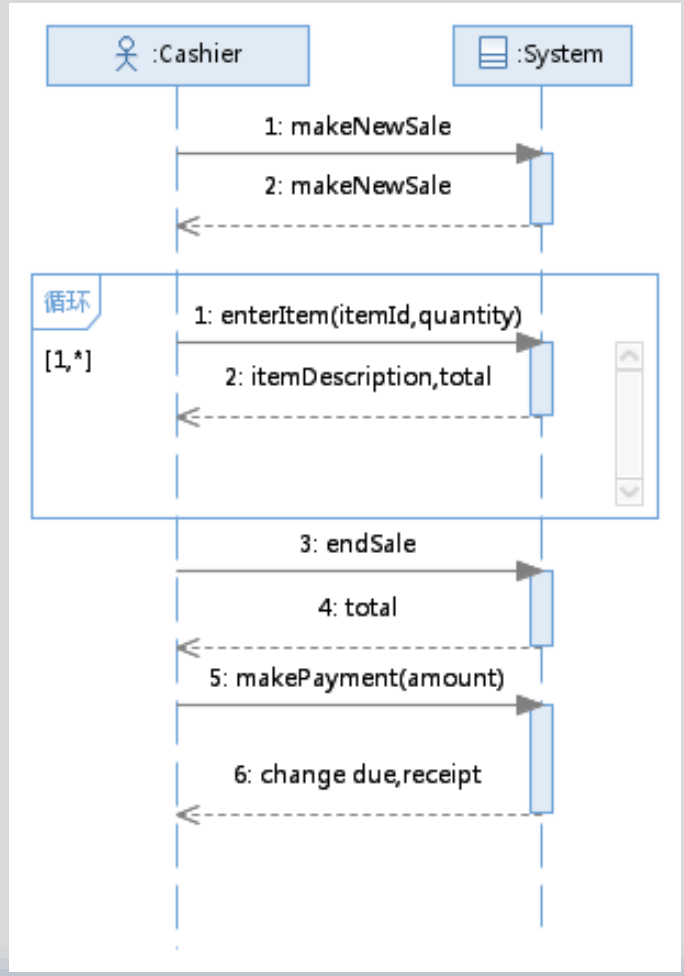
信息专家(Information Expert)模式

- 给对象分配职责的通用原则：将职责分配给拥有履行职责所必需信息的类，即信息专家。换言之，对象具有处理自己拥有信息的职责或能力。
- 根据信息专家模式，应该找到拥有履行职责所必须的信息的类，选取类的方法：
  - 如果在设计模型中存在相关的类，先到设计模型中查看；
  - 如果在设计模型中不存在相关的类，则到领域模型中查看，试着应用或扩展领域模型，得出相应的设计类。
- 职责的实现（即功能）需要信息，而信息往往分布在不同的对象中，一个任务可能需要多个对象（信息专家）协作来完成。

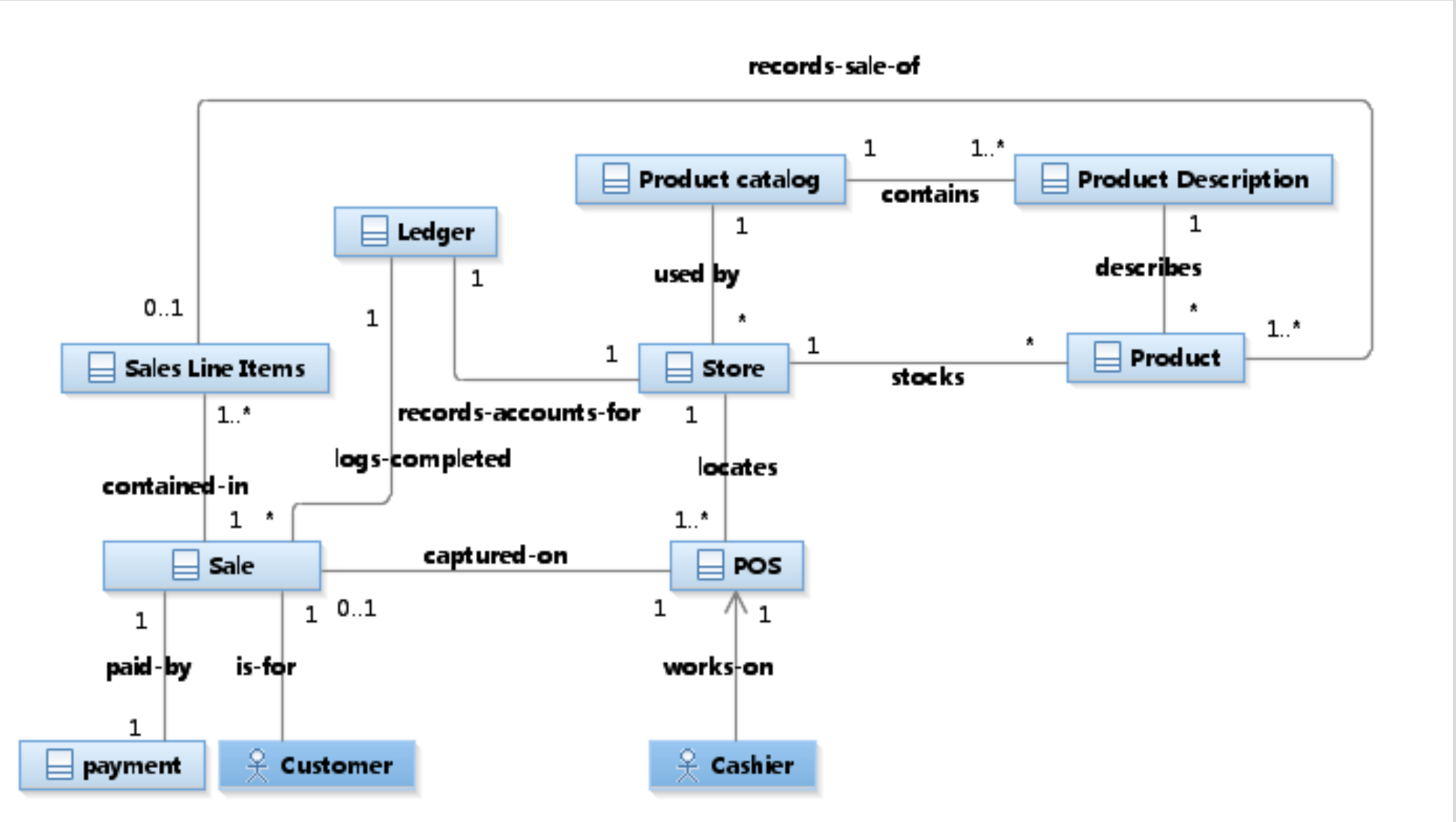
POS 机实例

- 超市柜台收银系统，简称 POS机系统
- 已知条件：
  - 角色：收银员
  - 用例：处理销售和处理支付；
  - SSD
  - 操作契约
  - 领域模型

成功场景：
1.顾客携带商品到达POS机收费口
2.收银员开始一次新的销售
3.收银员输入商品标识
4.系统记录单件商品，并显示该商品的描述、价格、累加值。
收银员重复3～4步，直到商品输入结束。
5.系统显示总值并计算税金。
6.收银员请顾客付款。
7.顾客支付，系统处理支付。
8.系统记录完整的销售信息，并将销售和付款信息发送到外部的帐务系统。
9.系统打印收据
10.顾客带着商品和收据离开。



POS 机实例



POS 机实例 设计步骤_1

- 选择系统架构
  - 基于B/S 的分层架构，要求至少能体现控制器层和应用逻辑层；
- 回顾并分析需求模型
  - 根据用例模型，主要的用例是：处理销售用例
  - 根据SSD得到四个系统事件：
    - makeNewItem
    - enterItem
    - endSale
    - makePayment
- 下一步设计：根据选择的系统架构，使用UML的交互图，为该用例的每一个系统事件确定对应的软件对象，并根据操作契约确定对象之间的关系

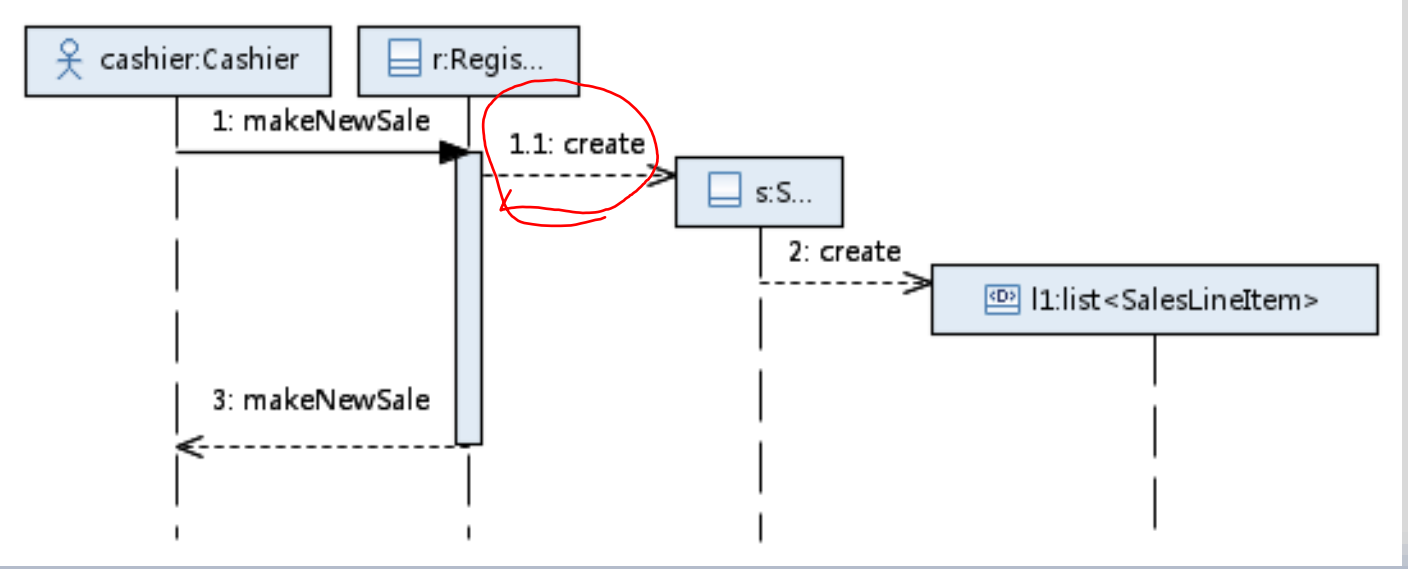
POS 机实例 设计步骤_2

- 对象设计：makeNewSale
- 操作契约：
  - 创建一个Sale实例s（实例创建）
  - s和Register建立关联（关联形成）
  - 初始化s的属性（属性修改）

问题：如何为系统操作makeNewSale选择控制器
解决方案：根据“控制器”模式，要么使用“外观控制器”，要么使用“用例控制器”。在本案例中，由于只存在少量的系统操作，为此选用“外观控制器”，以Register作为设计模型中的软件对象。
注意：此时的Register已经不再是物理终端设备

对象设计 MakeNewSale()

- 设计用例实现过程
  - Register 作为控制器对象接收系统事件 makeNewSale
  - 根据操作契约的第一条：创建一个Sale实例s，则Register具有创建s的职责；
  - Sale实例是Register创建，则形成了缺省的关联关系；
  - 操作契约：初始化s，即该实例能够记录多个销售的商品SalesLineItem，根据领域模型的关系Sale实例负责创建SalesLineItem的一个数组或集合；



对象设计 enterItem()

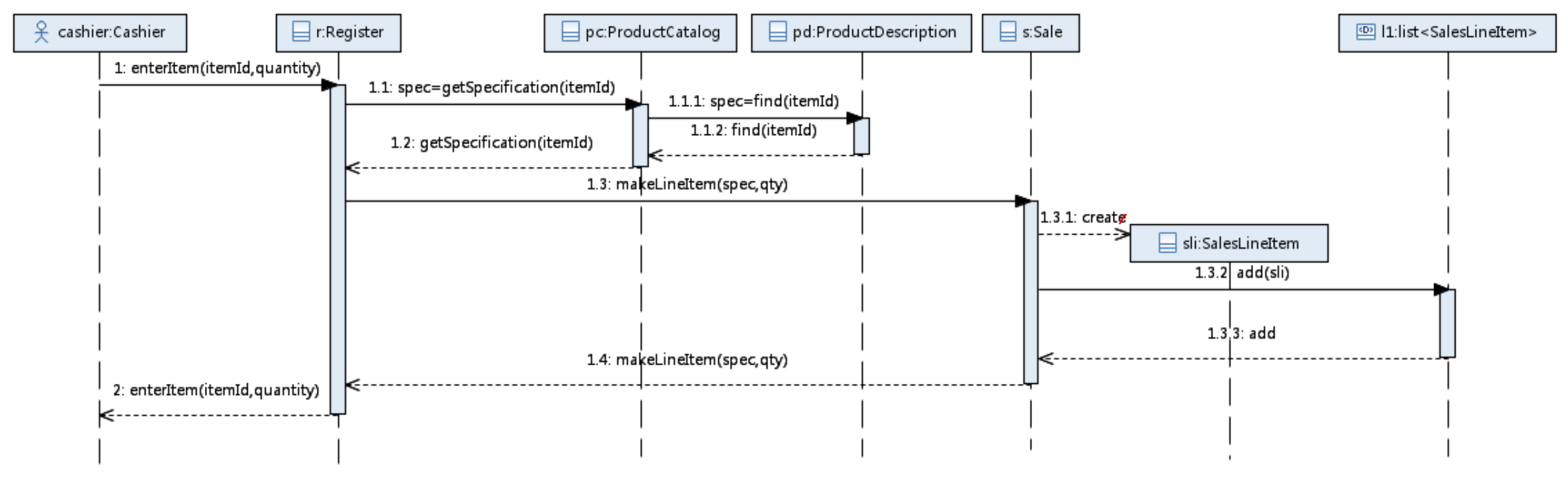
- 操作契约
  - 创建一个SalesLineItem实例sli（实例创建）
  - sli和当前的Sale建立关联（关联形成）
  - sli.quantity变成参数quantity（属性修改）
  - 实例sli在itemID匹配的基础上与ProductSpecification建立关联（关联形成）

问题1：控制器类的选择
问题2：是否要显示商品的描述和价格?
问题3：创建新的SaleLineItem
问题4：寻找ProductDescription

对象设计 enterItem()

- 问题_1：该系统事件属于同一个用例，为此控制器对象无论是外观还是用例控制器，都应该是Register；
- 问题_2：显然是需要在扫描物品时根据物品id找到对应的物品信息，关键问题是哪个对象负责查找?
  - 方案一：Register
  - 方案二：Sale
- 问题_3：根据领域模型可以确定是Sale对象负责创建SalesLineItem实例sli；
- 问题_4：根据领域模型可知ProductCatalog与ProductDescription相关联

方案一：enterItem()交互图

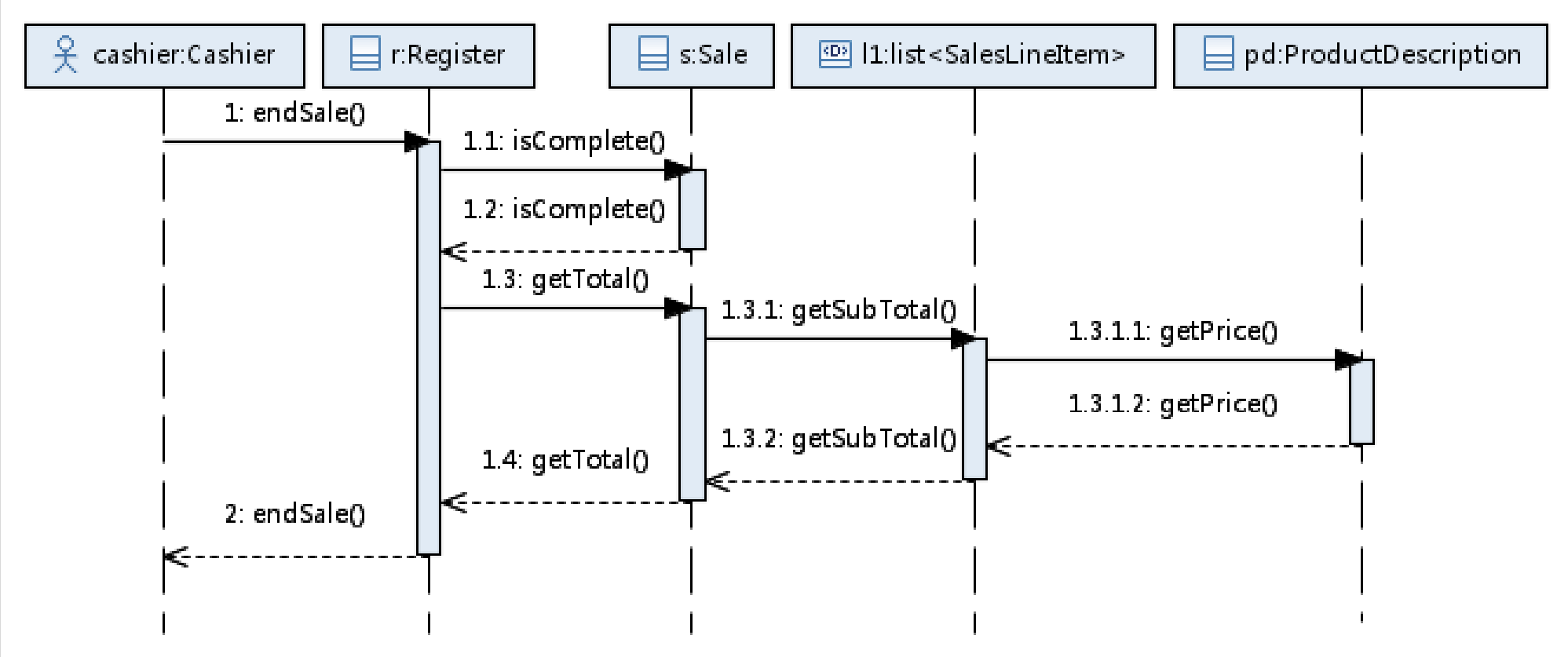


- 如果是方案二?
- 如果不使用数据对象list，而改用循环片段该如何表示?

对象设计endSale()

- 操作契约
  - s的属性 isComplete 被修改 = true;
  - 操作契约不完整，需要完善!
  - 意味着s 需要进行本次销售金额的计算：total = sum(subtotal=price*qty);
- 问题1：控制器对象?
- 问题2：哪个对象具有计算本次销售的金额（total）的职责?
- 问题3：哪个对象具有计算每种商品的销售金额（subtotal）的职责?
- 问题4：哪个对象具有每个saleLineItem的价格信息（price）？

endSale()交互图

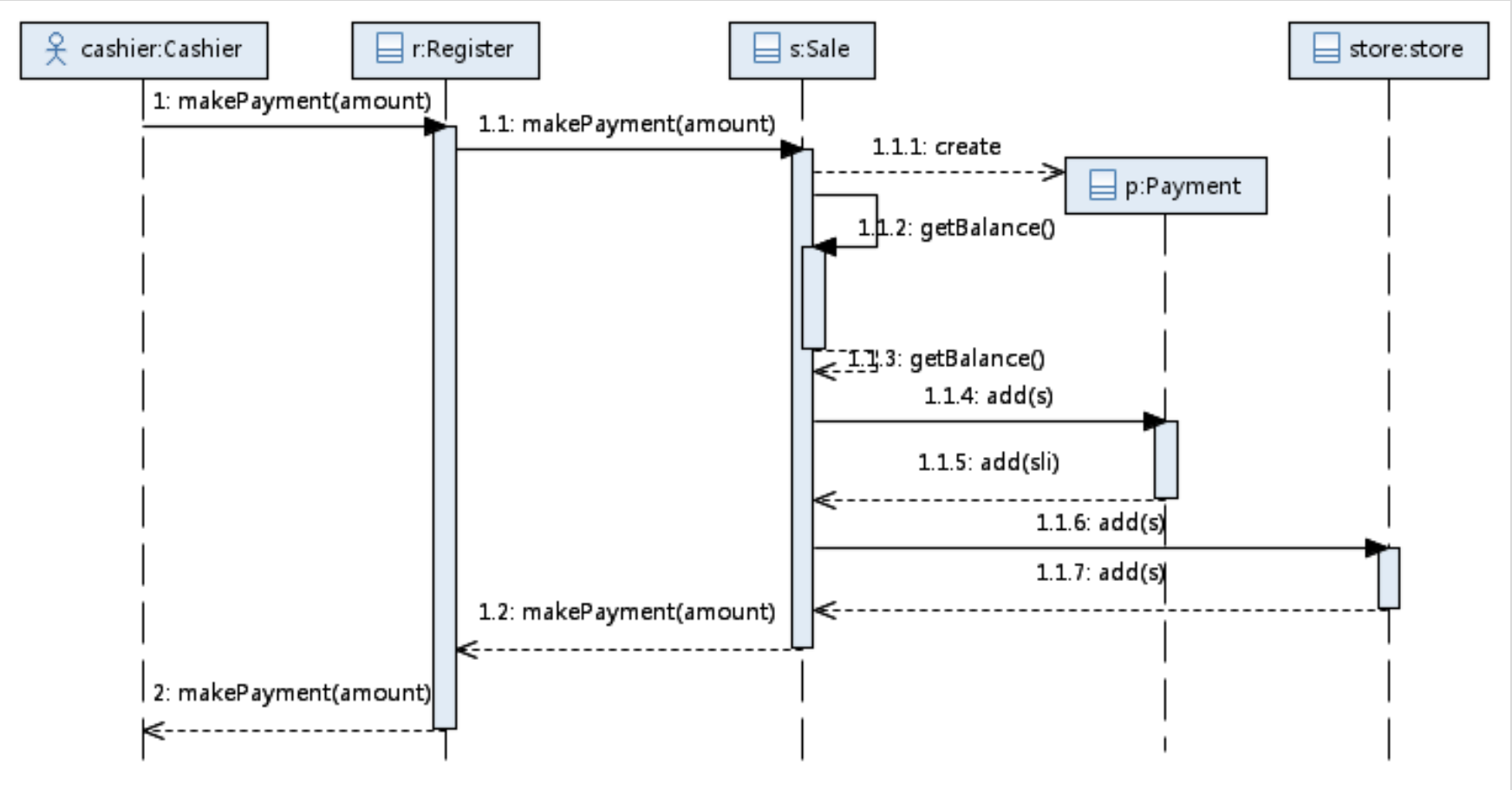


- 如果是方案二，此时的交互图应该是?

对象设计makePayment()

- 操作契约
  - 创建一个Payment实例p（实例创建）
  - 本次支付金额amount的修改（属性修改）
  - p和当前的Sale建立关联（关联形成）
  - 当前的Sale和Store建立关联（关联形成）：目的是向已完成销售的历史日志中添加。
- 问题1：p实例哪个对象负责创建?
- 问题2：计算找零金额=amount-total哪个对象应具有该职责?
- 问题3：哪个对象负责添加本次销售的记录或日志?
  - 方案一：Register
  - 方案二：Sale

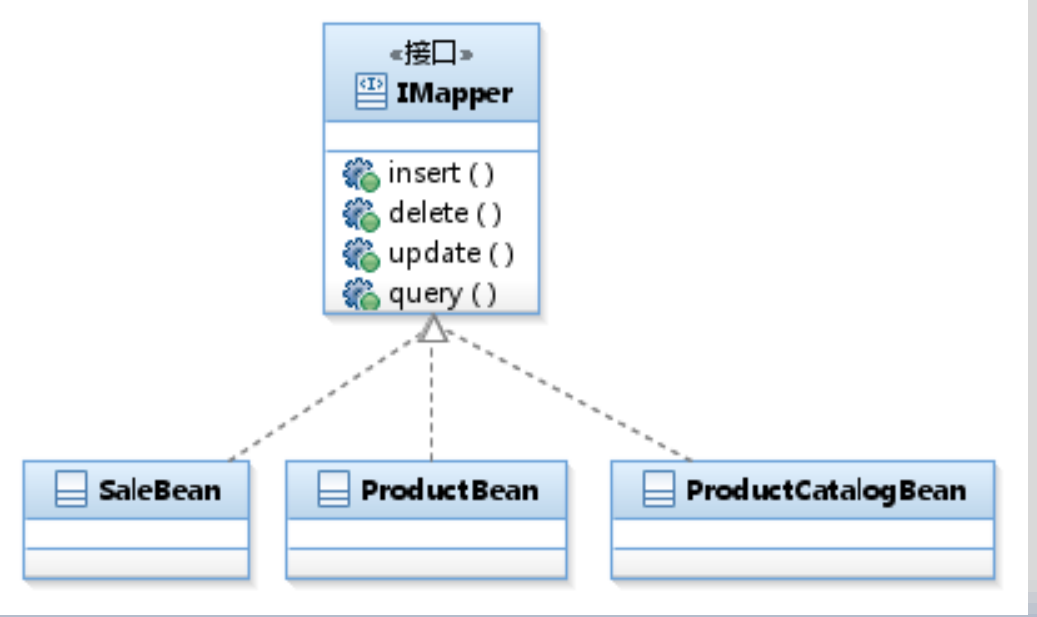
makePayment()交互图



- 如果要统计Register 的使用率，该如何设计?

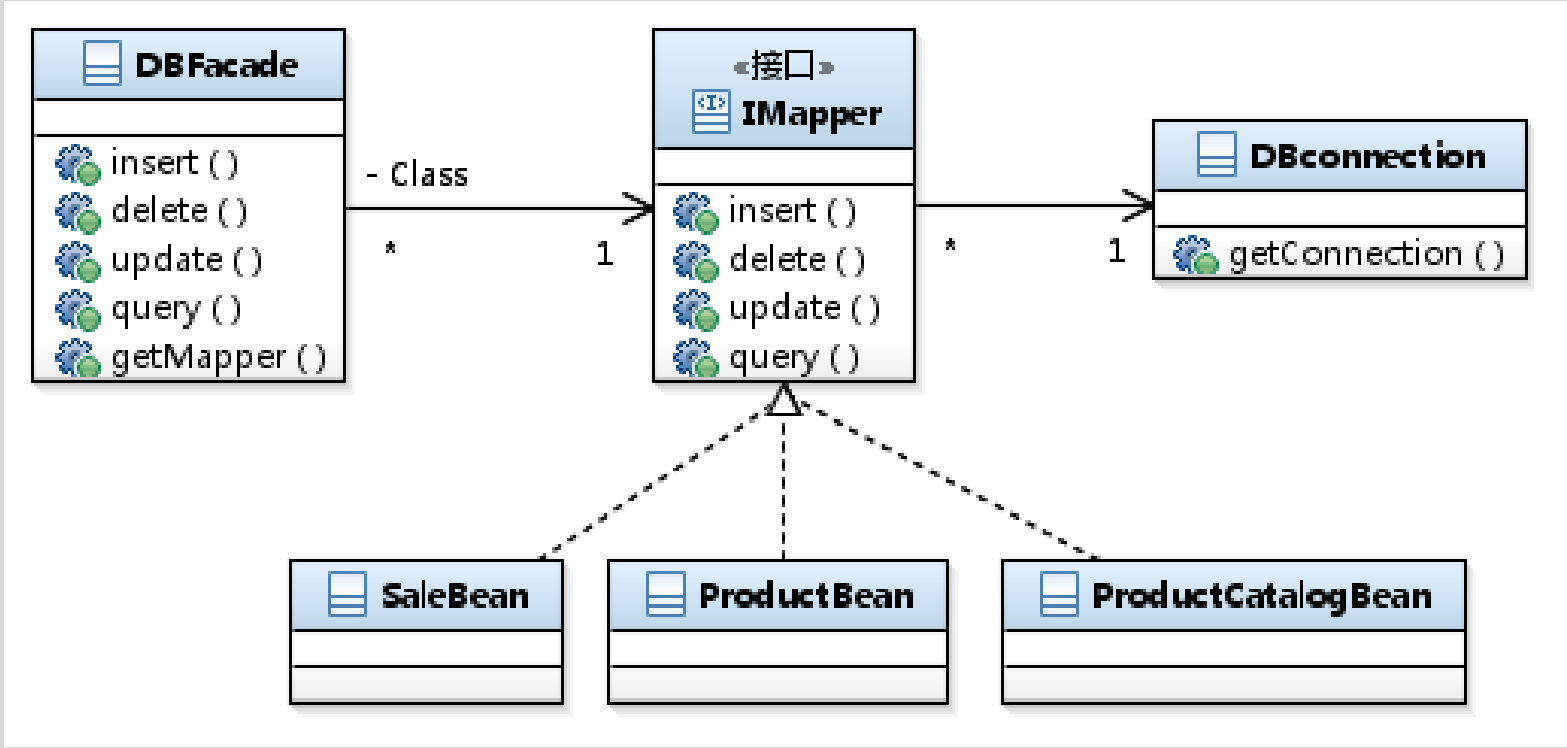
持久化层设计

- 问题一：哪个对象负责数据持久化?
- 为每一个领域对象设计一个专门负责其持久化的类：
  - SaleBean：负责销售信息持久化
  - ProductBean：负责商品信息持久化
  - ProductCatalogBean：负责商品目录信息持久化 ……
- 以上类中都包含增加、修改、删除和查询操作，所以可以抽象出共同的接口：
  - IMapper /IBean



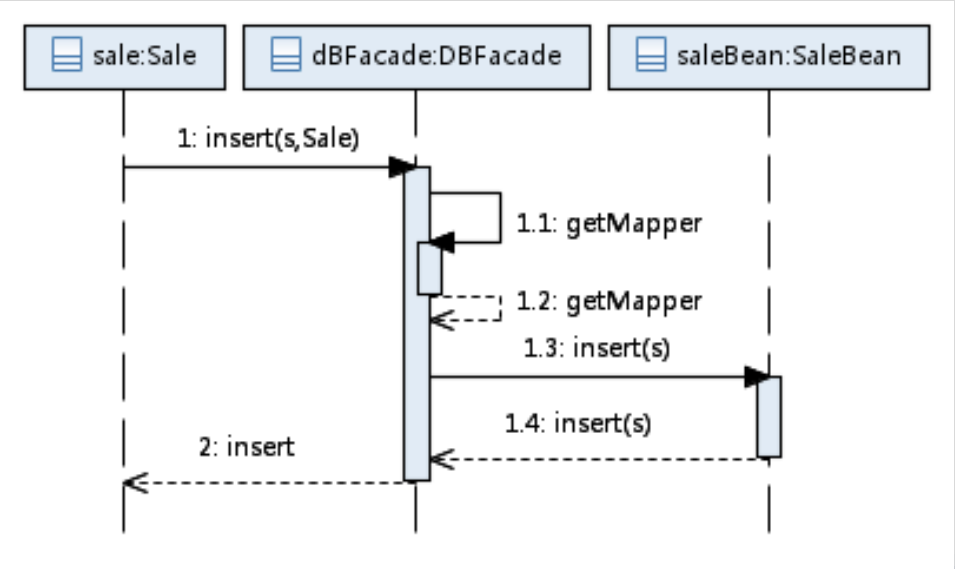
DBFacade类设计

- 问题二：哪个对象负责维护领域类和负责其持久化的对应Mapper类之间的对应关系？
- 设计类DBFacade



销售信息持久化过程

- Sale 对象实例化Sale类得到对象s；
- 调用持久化层DBFacade对象的insert操作，即向DBFacade发送insert消息；
- insert操作调用DBFacade对象的getMapper来获取负责对类Sale的对象进行持久化的对象SaleBean；
- insert操作调用SaleBean对象的insert操作，让其将s对象持久化到存储介质。



创建设计类图

- 通过类职责分配，找出了实现用例的类，以及类的职责。结合分析阶段的领域模型，可以得到设计阶段的类图，简称设计类图。
- 设计类图中主要定义类、类的属性和操作，但是不定义实现操作的算法。

1、通过扫描所有的交互图以及领域模型中涉及的类，识别软件类。

2、将领域模型中已经识别出来的部分属性添加到类中。

3、根据交互图为软件类添加方法。忽略软件类的构造函数和get/set方法；

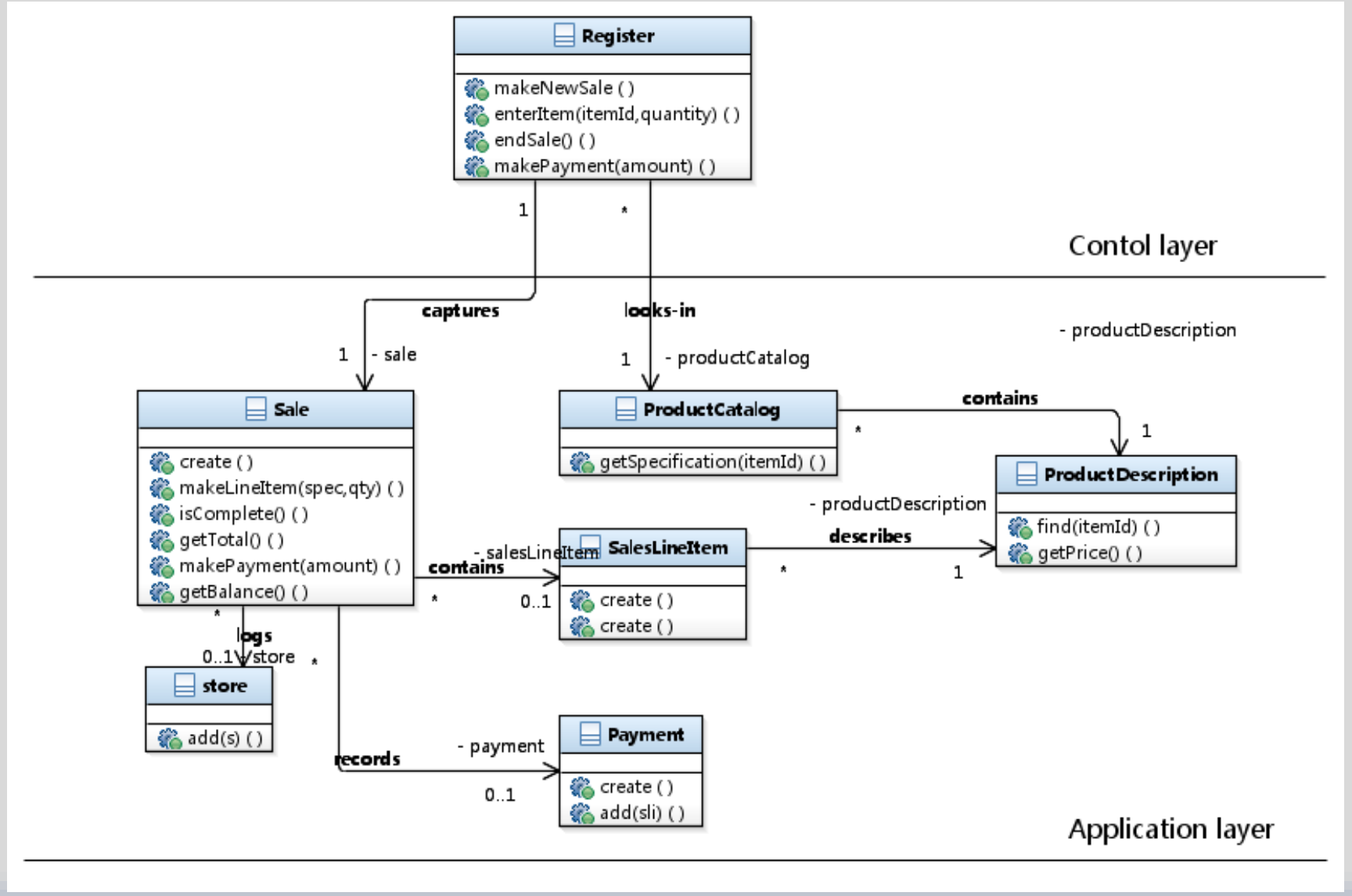
4、添加更多的类型信息。包括属性类型、方法参数类型以及返回类型。

5、添加关联和导航。定义A到B带导航修饰关联的常见情况有以下几种：

- A发送一个消息到B；
- A创建一个B的实例；
- A需要维护到B的一个连接。

6、类成员的细节表示（可选）。如成员的属性可见性，方法体的描述等。

控制器/处理层设计类图

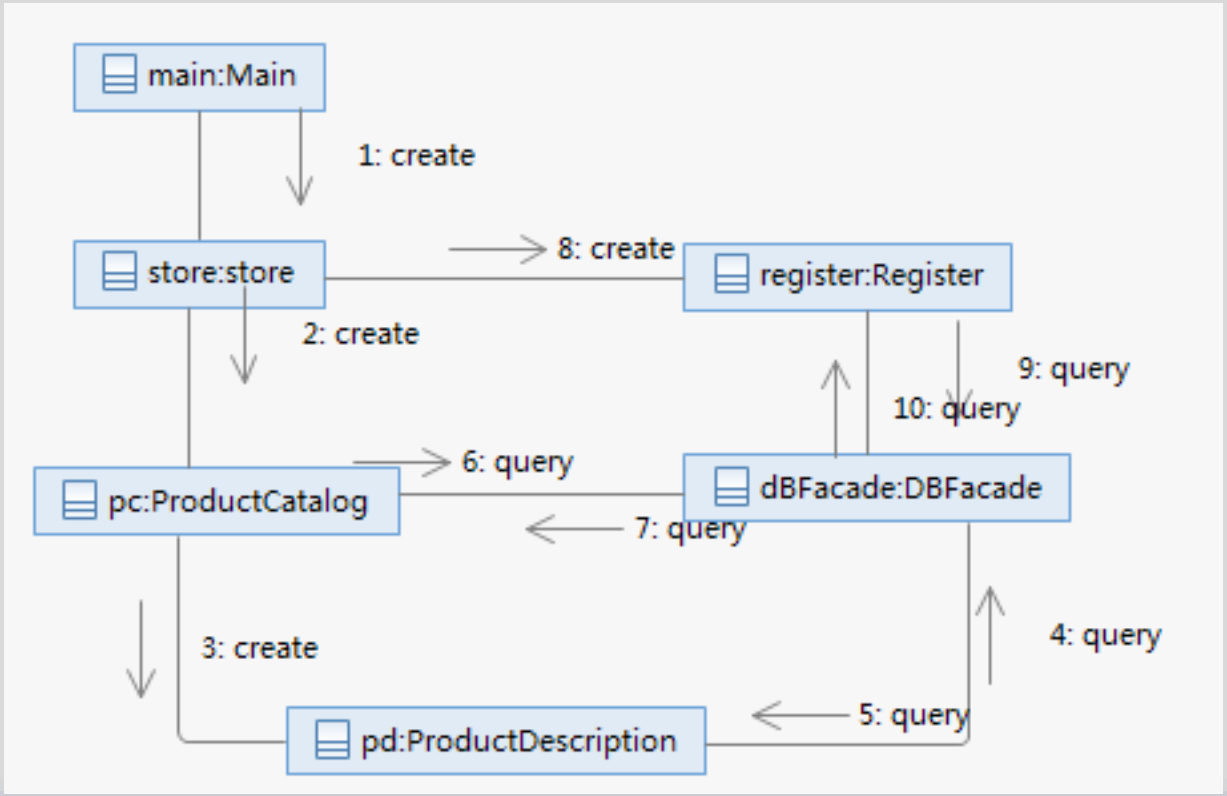


startUp设计

- 大多数系统在启动过程中都要做一些系统初始化操作，通过设置一个startUp功能来完成相关初始化工作。
- startUp的实现：
  - 需要初始化的领域对象有哪些?
    - 具有组合或聚合关系的根类；
    - 具有持久化保存的信息类；
    - 控制器类；
  - 软件服务如何启动?
    - 建立一个初始领域对象，由它负责后续直接领域对象的创建。
    - 应用发送create消息以创建初始领域对象
    - 如果初始领域对象控制进程，则应用继续发送run消息给初始领域对象，移交应用控制权。

POS机 的Store.startup

- 根据以上原则和分析结果，可有如下初始化内容：选择Store对象为根对象
  - 创建Store, Register, ProductCatalog, ProductDescription
  - 建立ProductCatalog, ProductDescription关联
  - 建立Store与ProductCatalog的关联
  - 建立Store与Register的关联
  - 建立Register 与ProductCatalog



动态结构设计总结

- 动态结构设计
  - 输入条件：用例，SSD，操作契约及领域模型
  - 用例实现过程：
    - 1、选择某一用例；
    - 2、查看该用例的SSD，选择某一指令；
    - 3、查看该指令对应的操作契约；
    - 4、结合已经确定的软件架构，设计并确定该指令进入系统后各层次的软件对象及其交互；
    - ……
    - 当所有SSD中的指令对应的交互图都已完成，结束该用例的实现过程设计
  - 输出结果：用例的一系列交互图，展示并证明系统如何执行用例的过程
    - 确定了用例对应的软件对象及其交互；
    - 确定了每个软件对象在该用例中必须具备的方法；

用例级别 静态结构设计总结

- 输入条件：用例的动态结构
- 静态结构整理步骤：
  - 1、根据已确定的软件架构层次；
  - 2、扫描每个交互图中某一软件层次中已确定的软件对象；
    - 多个交互图中重复出现的软件对象，在静态结构图中只保留一个；
    - 确定该层次中多个软件对象之间是否有交互；
  - 3、扫描其他层次的软件对象；
  - 4、确定层次之间软件对象的交互关系（以定向关联表示）；
- 输出结果：用例级别的静态类图
  - 确定每个软件类的属性及类型定义；
  - 确定每个软件类的方法及参数定义；
- ……
- 系统级静态结构：对每个用例的静态类图进行扫描，去除重复出现的软件类，修改并确定软件类层次之间以及同层软件类之间的关系