交互图

内容

- 概述
- 顺序图
- 通信图
- 定时图
- 交互概述图
- 职责分配模式

内容

- 概述
- 顺序图
- 通信图
- 定时图
- 交互概述图
- 职责分配模式

交互图

- 交互图显式地描述对象如何交互以提供特定的系统行为。

交互图

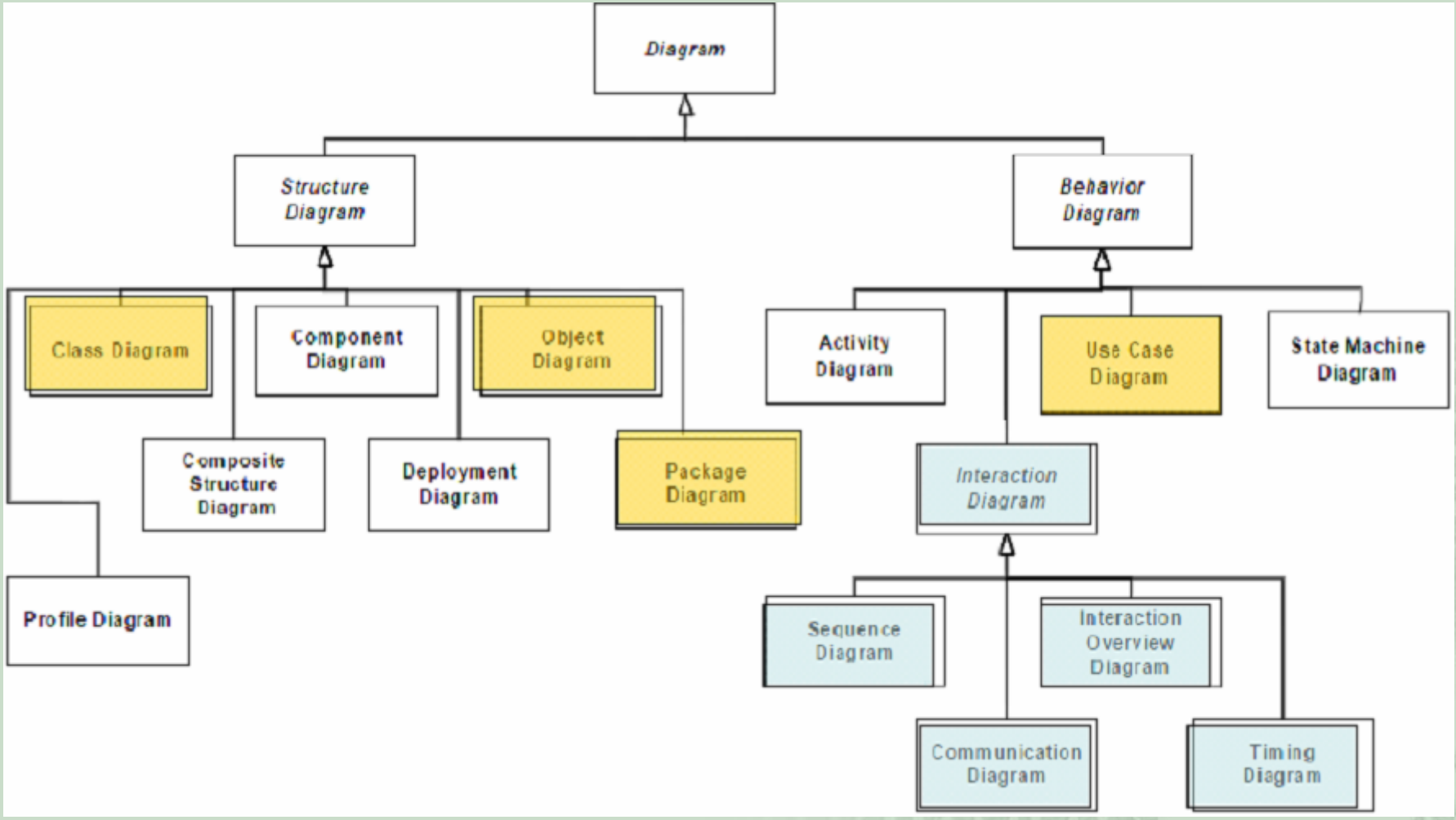


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
Profile Diagram
Sequence Diagram
Interaction Overview Diagram
Communication Diagram
Timing Diagram

UML交互图的类型

- Sequence Diagram
  顺序图（序列图）
- Communication Diagram
  通信图（协作图）
- Timing Diagram
  定时图（时序图）
- Interaction Overview Diagram
  交互概述图

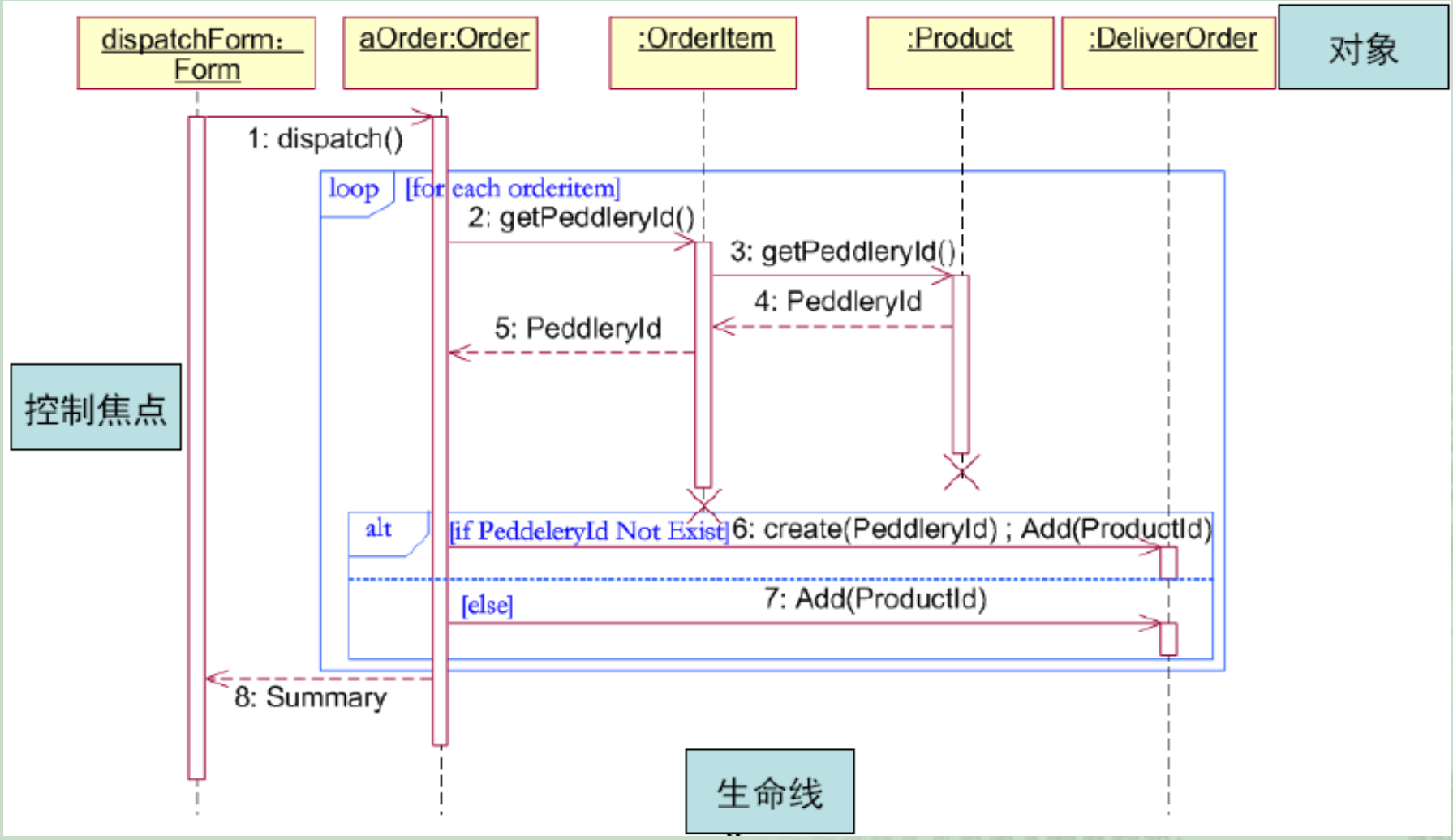
内容

- 概述
- 顺序图
- 通信图
- 定时图
- 交互概述图

顺序图定义

- 顺序图显示参与交互的对象及对象之间消息交互的顺序。

顺序图


dispatchForm:
Form
aOrder:Order
:OrderItem
:Product
:DeliverOrder
对象
1: dispatch()
loop
[for each orderitem]
2: getPeddleryId()
3: getPeddleryId()
4: PeddleryId
5: PeddleryId
控制焦点
alt
[if PeddeleryId Not Exist]
6: create(PeddleryId) ; Add(ProductId)
[else]
7: Add(ProductId)
8: Summary
生命线

顺序图

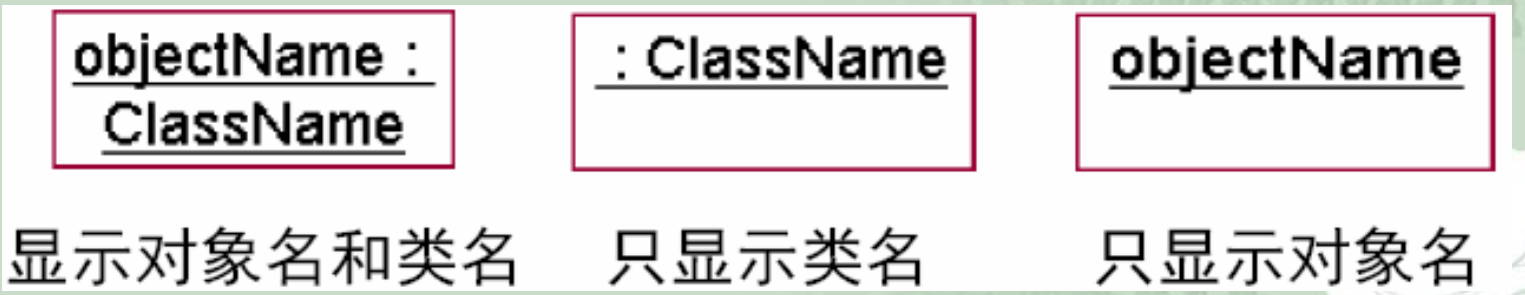
- 顺序图是一个二维图形
- 在顺序图中水平方向为对象维，沿水平方向排列参与交互的对象
- 竖直方向为时间维，沿垂直向下方向按时间递增顺序列出各对象所发出和接收的消息
- 水平轴上的对象间的相互顺序并不重要

顺序图的组成元素

- 对象（Object，包括Actor的实例）
- 生命线（Lifeline）
- 控制焦点（Focus of control）
- 消息（Message）

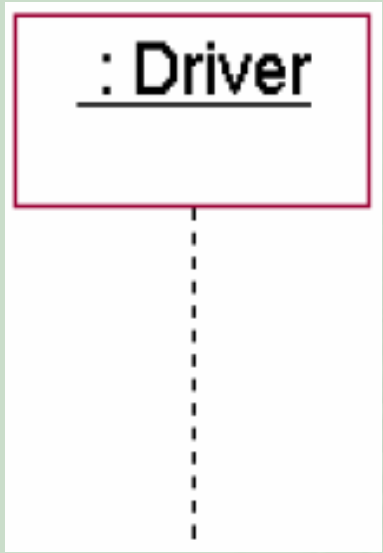
对象

- 顺序图中对象的符号和对象图中一样。
- 对象置于顶部意味着交互开始时对象就已经存在了；对象不置于顶部意味着对象是在交互过程中创建的。
- 注意三种命名方式



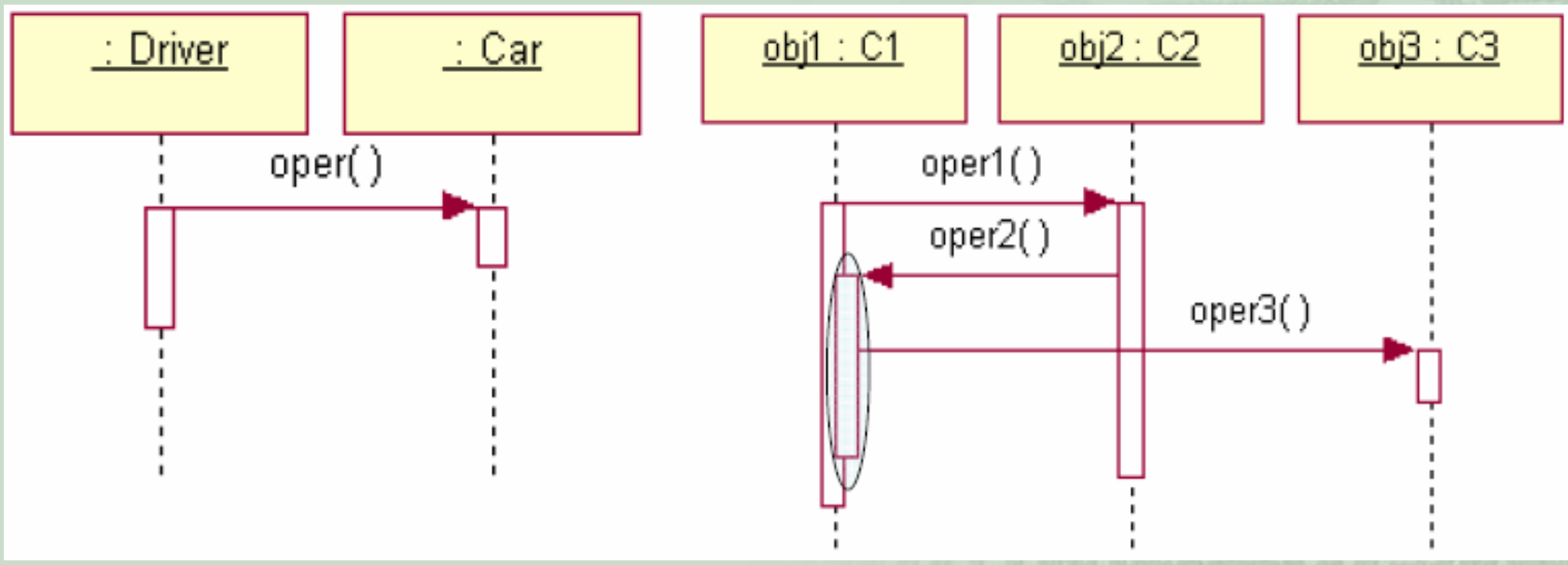
生命线（Lifeline）

- 生命线表示对象在一段时间内的存在。
- 在顺序图中生命线表示为从对象图标向下延伸的一条虚线。



控制焦点（Focus of control）

- 顺序图可以描述对象的激活和去激活。
- 用生命线上的小矩形表示处于激活或控制期，在这个时间段内， 对象执行相应的操作。



消息（Message）

- 消息描述对象之间某种形式的通信
- 消息用箭头表示，箭头的类型表示了消息的类型
- 消息在生命线上的位置并不是消息发生的准确时间，只是一个相对位置

消息的种类

- 调用
- 返回
- 发送
- 创建
- 销毁

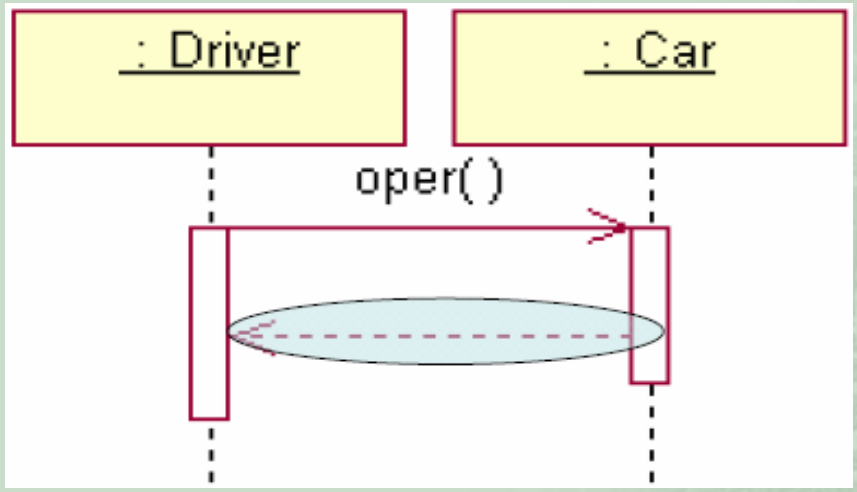
调用

- 最为常用的一种消息
- 表示调用某个对象的一个操作
- 可以是对象之间的调用，也可以是对对象本身的调用
- 表示方法：

- 可以在符号上加上顺序编号、消息名称和参数

返回

- 表示被调用的对象向调用者返回一个值
- 表示方法：在UML交互图中，采用虚线箭头线来表示，在箭头线上应标明返回值

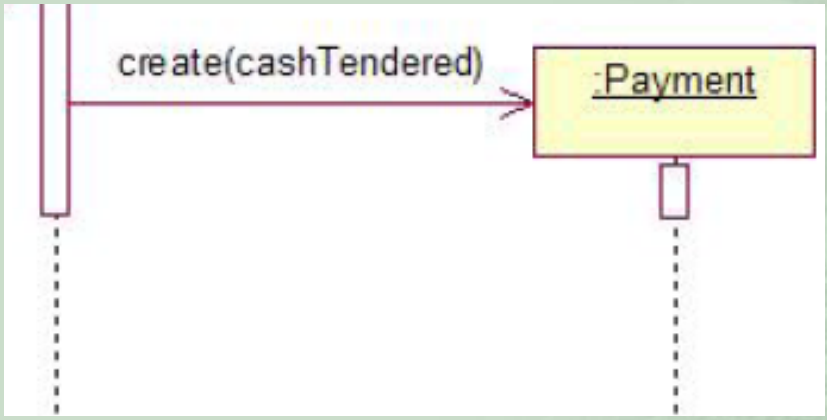


发送

- 发送是指向对象发送一个信号
- 信号和调用不同，它是一种事件，用来表示各对象间进行通信的异步激发机制
- 调用是一种同步机制，信号是一种异步机制

创建

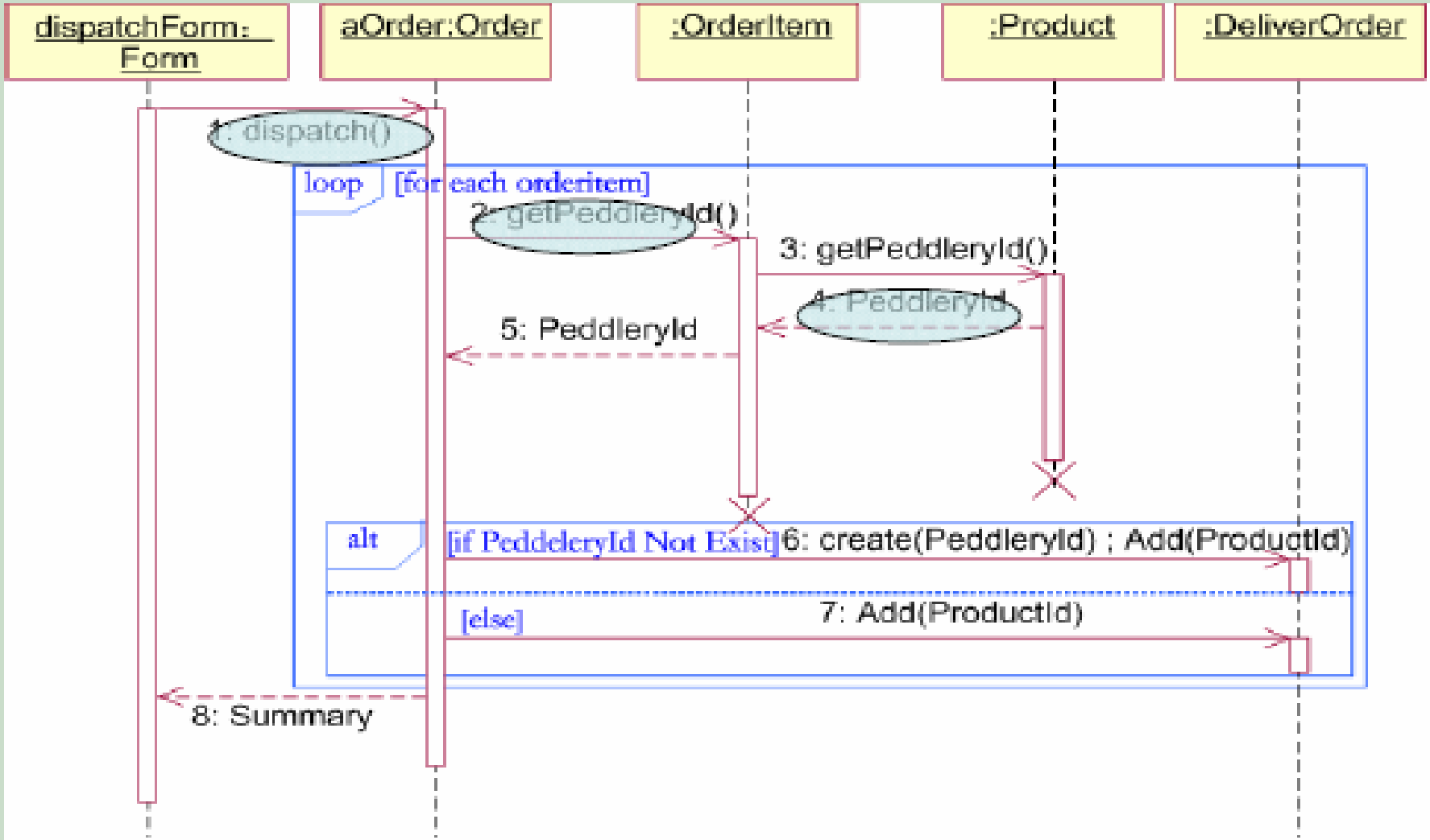
- 创建一个对象
- 创建对象通常通过构造函数（方法）来实现
- 对象创建之后，生命线即开始



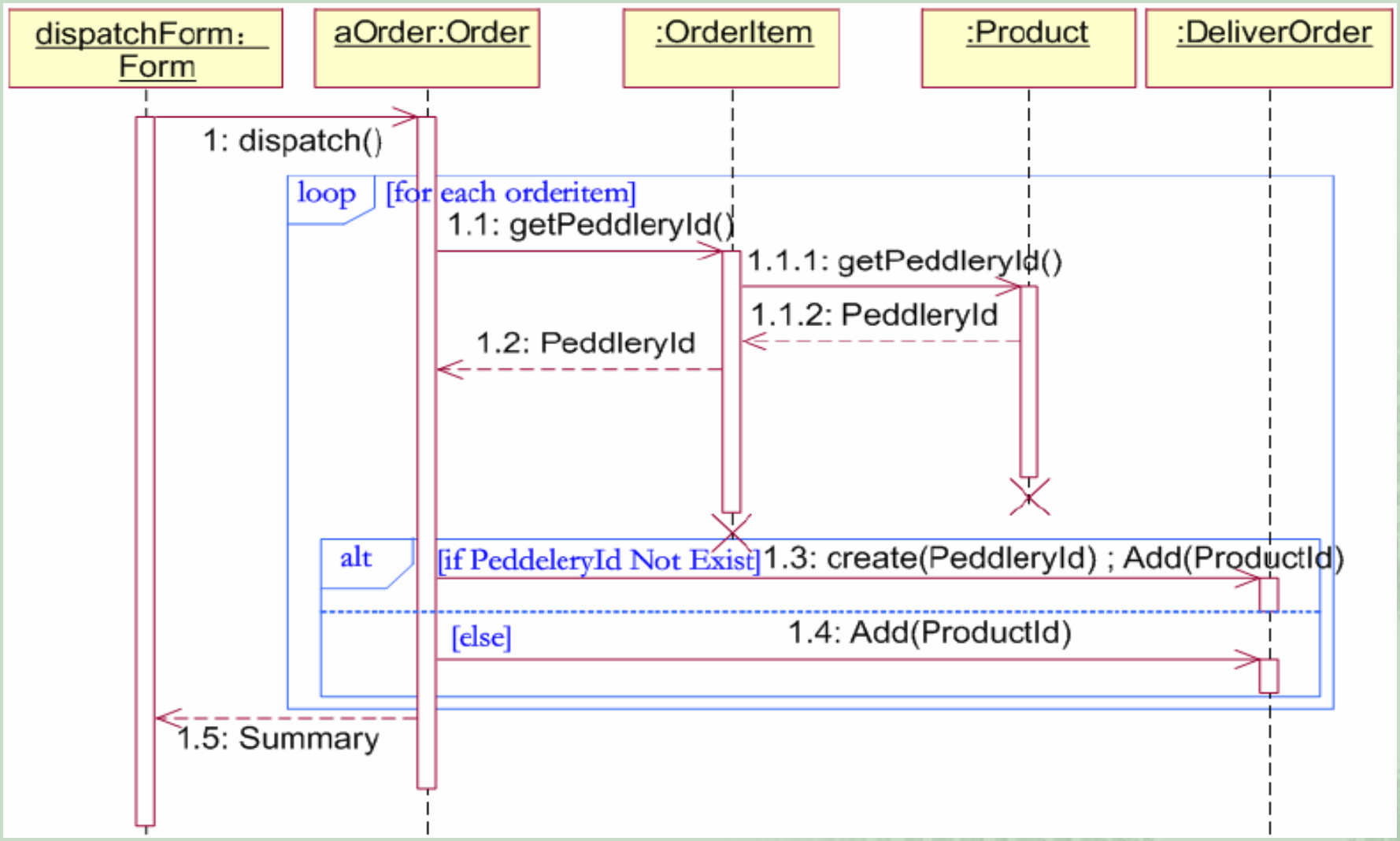
销毁

- 销毁一个对象
- 销毁一个对象一般是利用析构函数来实现
- 通常连接着的是目标对象的生命终止符号——一个较大的叉形符号

消息的编号：顺序编号


dispatchForm:
Form
aOrder:Order
:OrderItem
:Product
:DeliverOrder
1: dispatch()
loop
[for each orderitem]
2: getPeddleryId()
3: getPeddleryId()
4: PeddleryId
5: PeddleryId
alt
[if PeddleryId Not Exist]
6: create(PeddleryId) ; Add(ProductId)
[else]
7: Add(ProductId)
8: Summary

消息的编号：嵌套编号

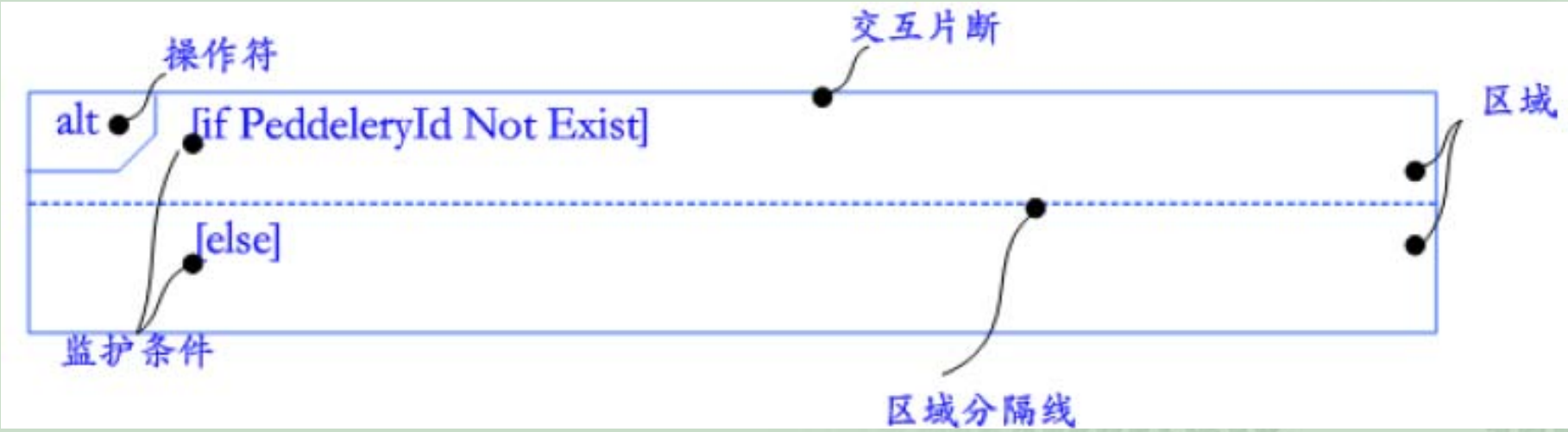

dispatchForm：Form
aOrder:Order
:OrderItem
:Product
:DeliverOrder
1: dispatch()
loop
[for each orderitem]
1.1: getPeddleryId()
1.1.1: getPeddleryId()
1.1.2: PeddleryId
1.2: PeddleryId
alt
[if PeddeleryId Not Exist]
1.3: create(PeddleryId) ; Add(ProductId)
[else]
1.4: Add(ProductId)
1.5: Summary

消息的语法格式

- [predecessor][guard-condition][sequence-expression] [return-value:=] message-name ([argumentlist])
  - predecessor：必须先发生的消息的列表
  - guard-condition：表示只有在条件满足时才能发送该消息
  - sequence-expression：消息顺序表达式
  - return-value：返回值
  - message-name：消息名称
  - argument-list：消息的参数列表

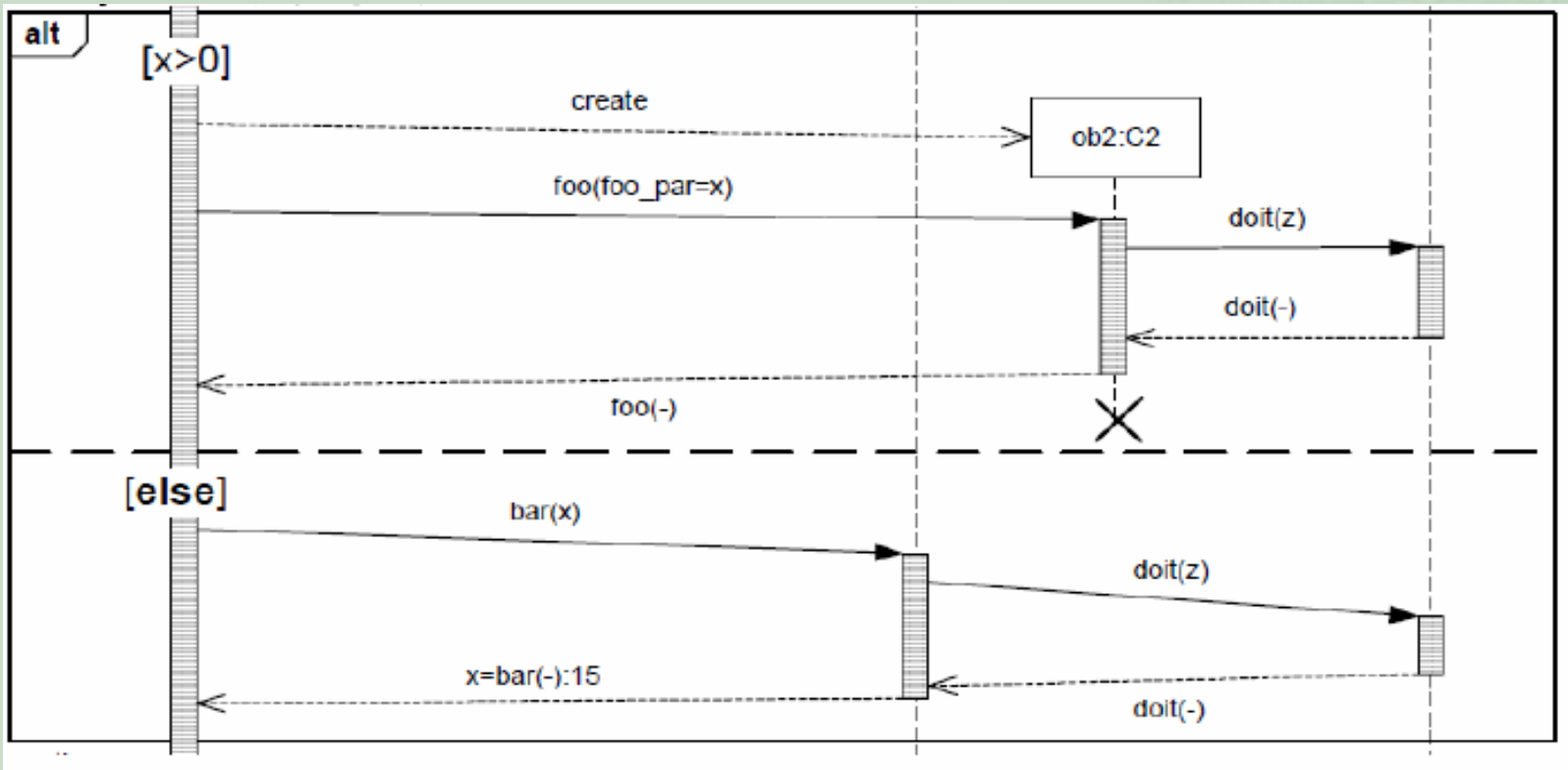
循环与分支的表达

- 引入交互片段、区域和操作符来表达



分支

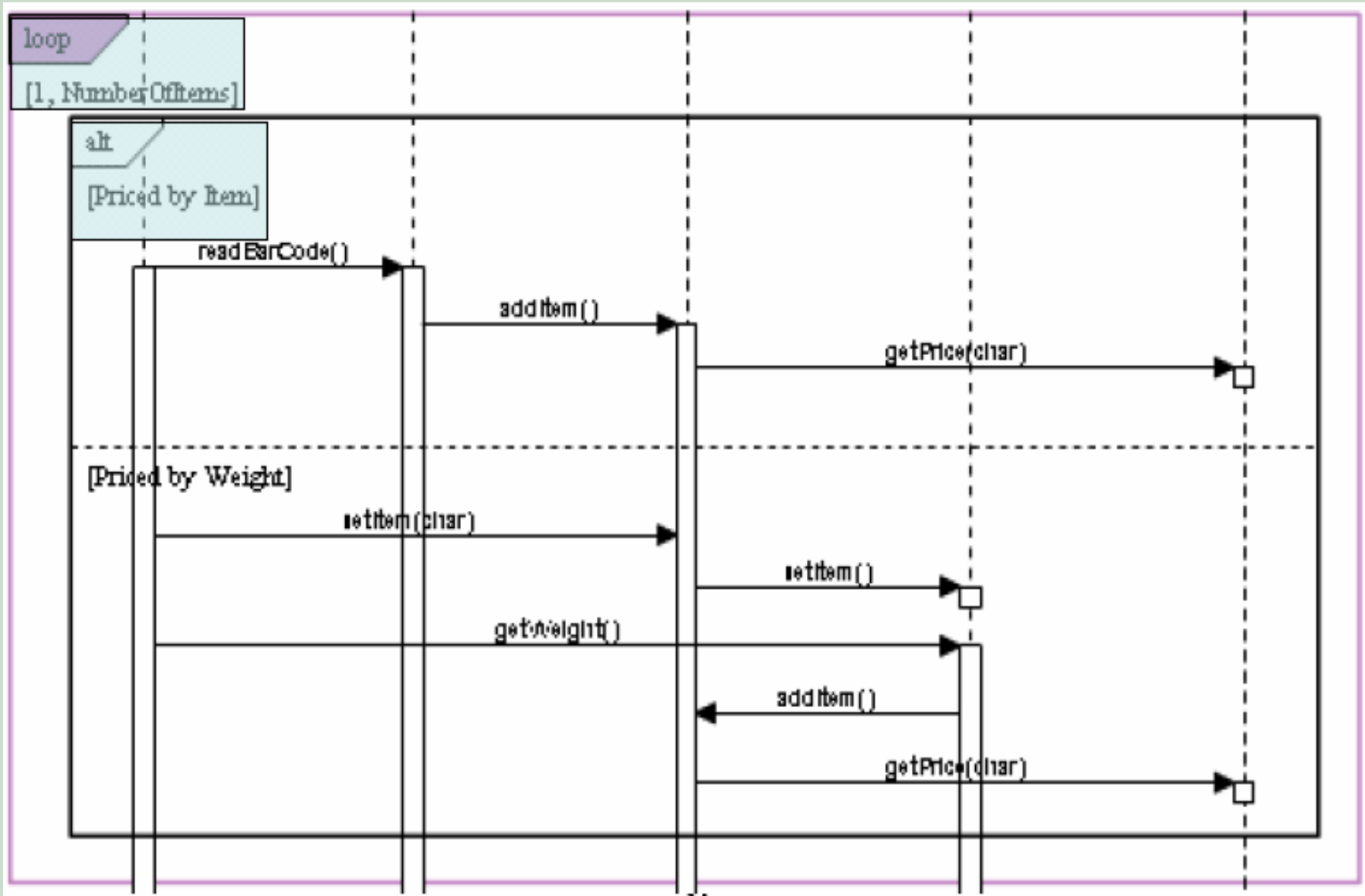
- 可以表示分支的操作符
  - alt:支持多条件
  - opt:支持单条件



循环

- 表示循环的操作符
- loop
- 说明该片段将可以执行多次，具体的次数由循环次数和监护条件表达式说明
- Loop(1,n):表示for i=1;i<n;i++
- Loop(10):表示执行10次

带有循环的交互区域示例



交互片段的一些操作符

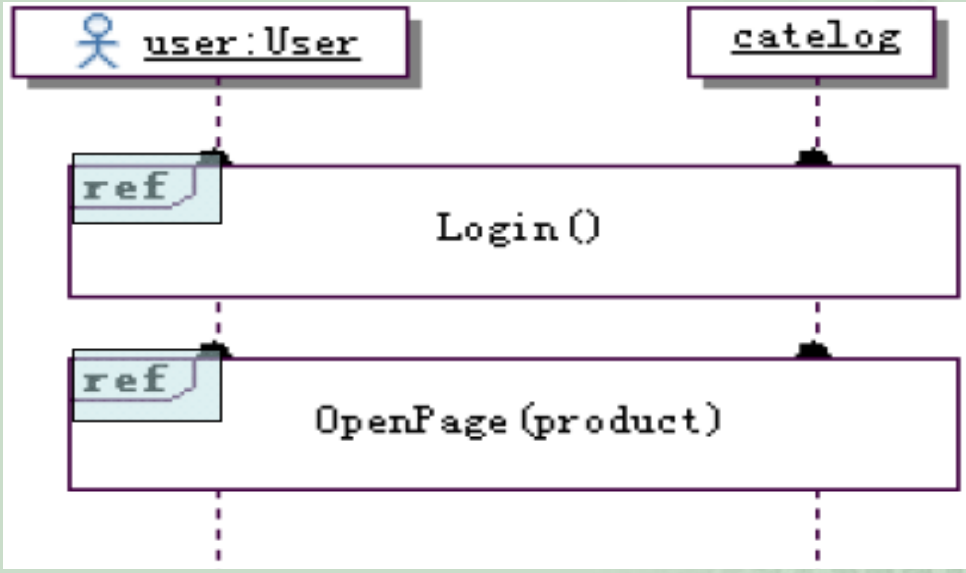
- assert：
  - 断言包含一个片断
- consider：
  - 包含一个子片断和一个消息类型列表
  - 只有列表中的消息类型可以出现在子片断中
- ignore
  - 与consider相反，会忽略列表中的消息类型

交互片段的一些操作符

- break
  - 跳转，中断，包含一个含有监护条件的子片断，如果监护条件为“true”，则执行子片段，而且不执行子片段后面的其它交互
  - 跳转最常用来做异常处理
- critical
  - 表示子片段是临界区域，通常用来表示一个原子性的连续操作，比如一些事务性操作

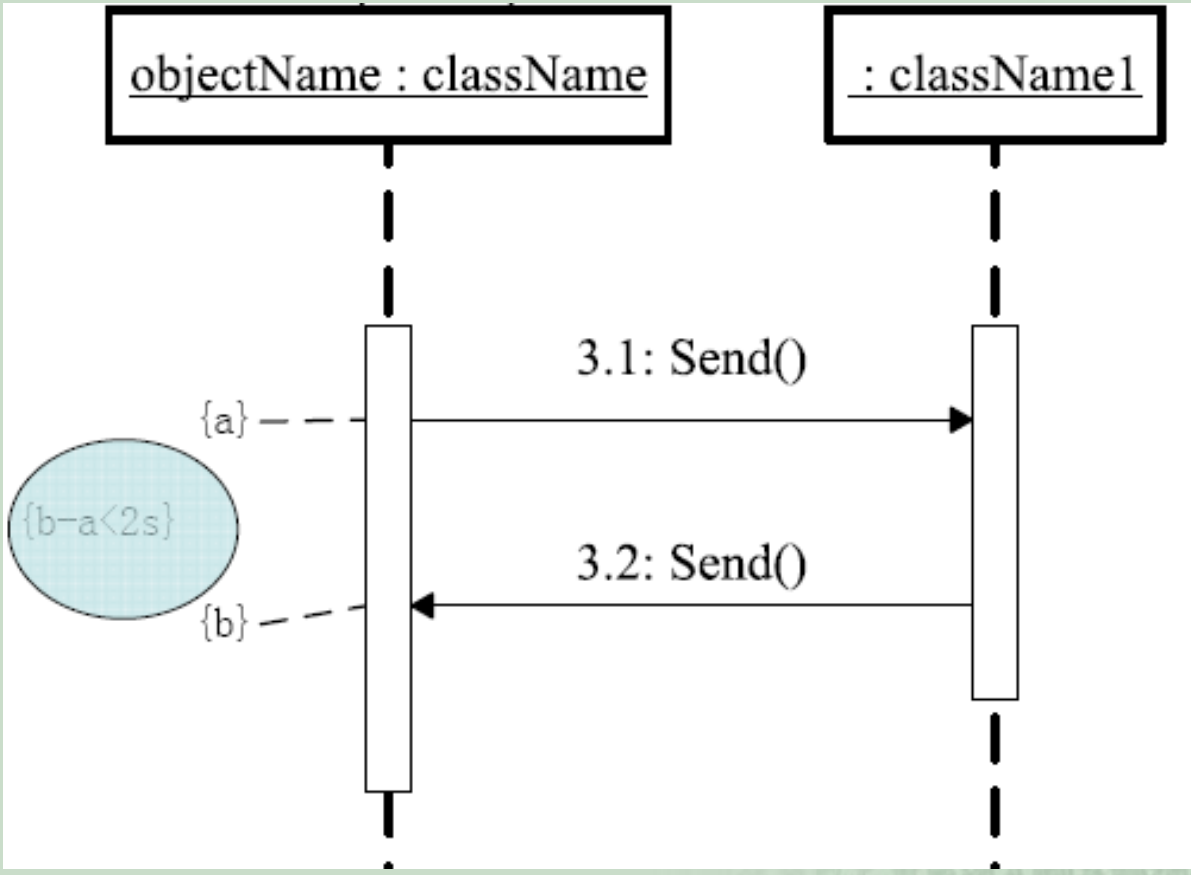
交互片段的一些操作符

- par
  - 表示包含两个或更多并发执行的子片断
- ref
  - 用来表示在一个交互图中引用其它的交互图



顺序图中的约束

- 用constraint(约束)来表示



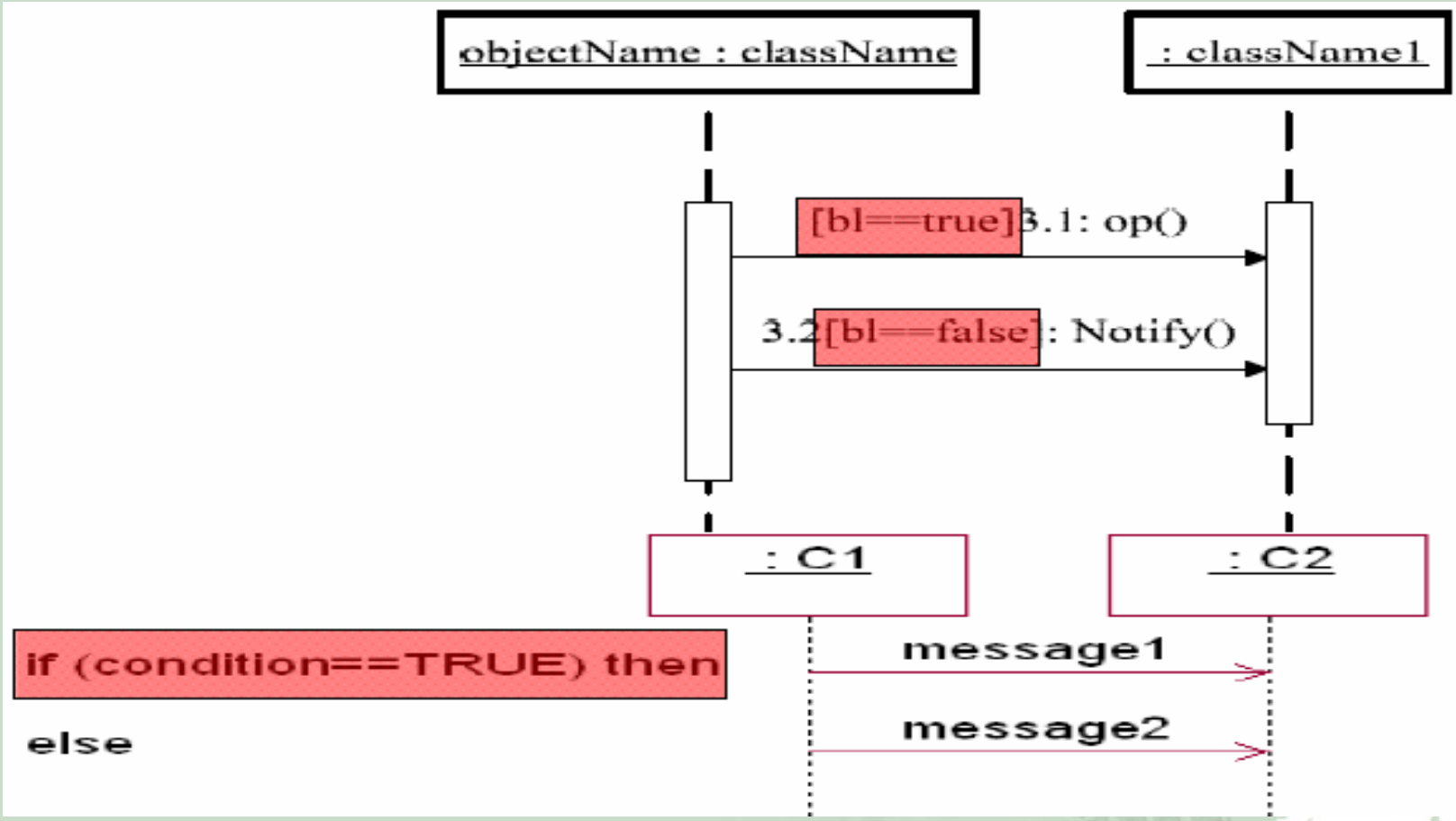
顺序图建模步骤

- 确定交互过程的上下文
- 识别参与交互过程的对象
- 为每个对象设置生命线，即确定哪些对象存在于整个交互过程中，哪些对象在交互过程中被创建和撤销
- 从引发这个交互过程的初始消息开始，在生命线之间从顶到下依次画出随后的各个消息
- 如果需要表示消息的嵌套或消息发生的时间点，则可采用控制焦点
- 如果需要说明时间约束，则在消息旁边加上约束说明
- 如需要，可以为每个消息附上前置条件和后置条件

顺序图常见问题

- 如何在顺序图中表示消息的条件发送？
  - 在消息上加监护条件
  - 在消息名前加条件子句
  - 使用文字说明
  - 分成多个顺序图（每分支一图）

顺序图常见问题


objectName : className
: className1
[bl==true]3.1: op()
3.2[bl==false]: Notify()
: C1
: C2
if (condition==TRUE) then
else
message1
message2

顺序图建模风格

- 尽量按从左到右的顺序排列消息
- 由于一般的阅读习惯是从左到右，因此建议在顺序图中从左上角开始一个消息流
- 当然，有时不可能让所有的消息从左到右排列，比如当两个对象相互调用时

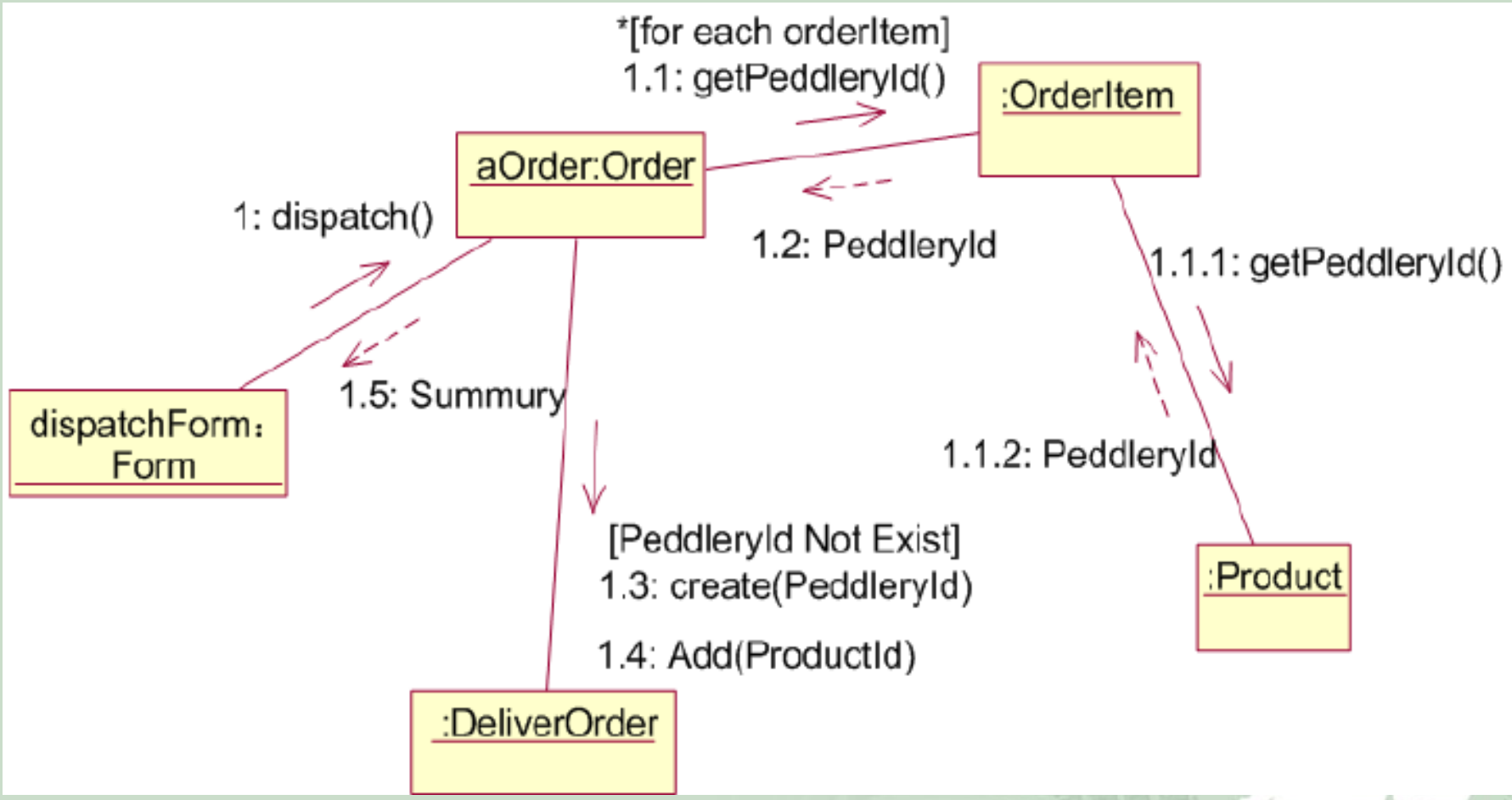
顺序图建模风格

- 当存在同一个类型的对象时，分别为它们命名，以保证清晰
- 把注意力集中在关键的交互上
- 不要包含无关的细节
- 例如：如果顺序图是用于描述业务逻辑的，就没必要包括对象和数据库之间的详细交互
- 保证建模的结果图足够简单，提高建模人员的工作效率，还可提高图的可读性

内容

- 概述
- 顺序图
- **通信图**
- 定时图
- 交互概述图
- 职责分配模式

通信图示例


*[for each orderItem]
1.1: getPeddleryId()
:OrderItem
aOrder:Order
1: dispatch()
1.2: PeddleryId
1.1.1: getPeddleryId()
dispatchForm：
Form
1.5: Summury
1.1.2: PeddleryId
[PeddleryId Not Exist]
1.3: create(PeddleryId)
:Product
1.4: Add(ProductId)
:DeliverOrder

通信图

- 通信图描述系统中对象之间通过消息进行的交互，强调参加交互的对象的组织结构
- 通信图的组成元素
  - 对象
  - 链接
  - 消息

链接

- 用来表示对象之间的语义连接
- 一般而言，链接是关联的一个实例

消息

- 通信图中的消息类型与顺序图中的相同
- 为了说明交互过程中消息的时间顺序，需要给消息添加顺序号
- 顺序号是在消息的前面加一个整数，每个消息都必须有唯一的顺序号

消息编号

- 无层次编号
- 嵌套编号

迭代标记

- 迭代实际上是用来表示循环的
- 用*号表示，通常还有迭代表达式，用来说明循环规则

迭代表达式	语义
[i:=1..n]	迭代n次
[I=1..10]	迭代10次
[while(表达式)]	表达式为true时才进行迭代
[until(表达式)]	迭代到表达式为true时为止
[for each(对象集合)]	在对象集合上迭代

监护条件

- 监护条件通常是用来表示分支的，也就是表示“如果条件为true，才发送消息”
- 在通信图中使用监护条件一定要有所限制，通常应只列出主要的监护条件，否则会影响其阅读。如果需要，尽可能还是通过顺序图来表示

通信图建模步骤

- 确定交互过程的上下文
- 识别参与交互过程的对象
- 确定对象之间的链接，以及沿着链的消息
- 从引发这个交互过程的初始消息开始，将随后的每个消息附到相应的链上
- 如果需要，消息编号可以表示为嵌套方式
- 如需要说明时间约束，则在消息旁边加上约束说明
- 如需要，可以为每个消息附上前置条件和后置条件

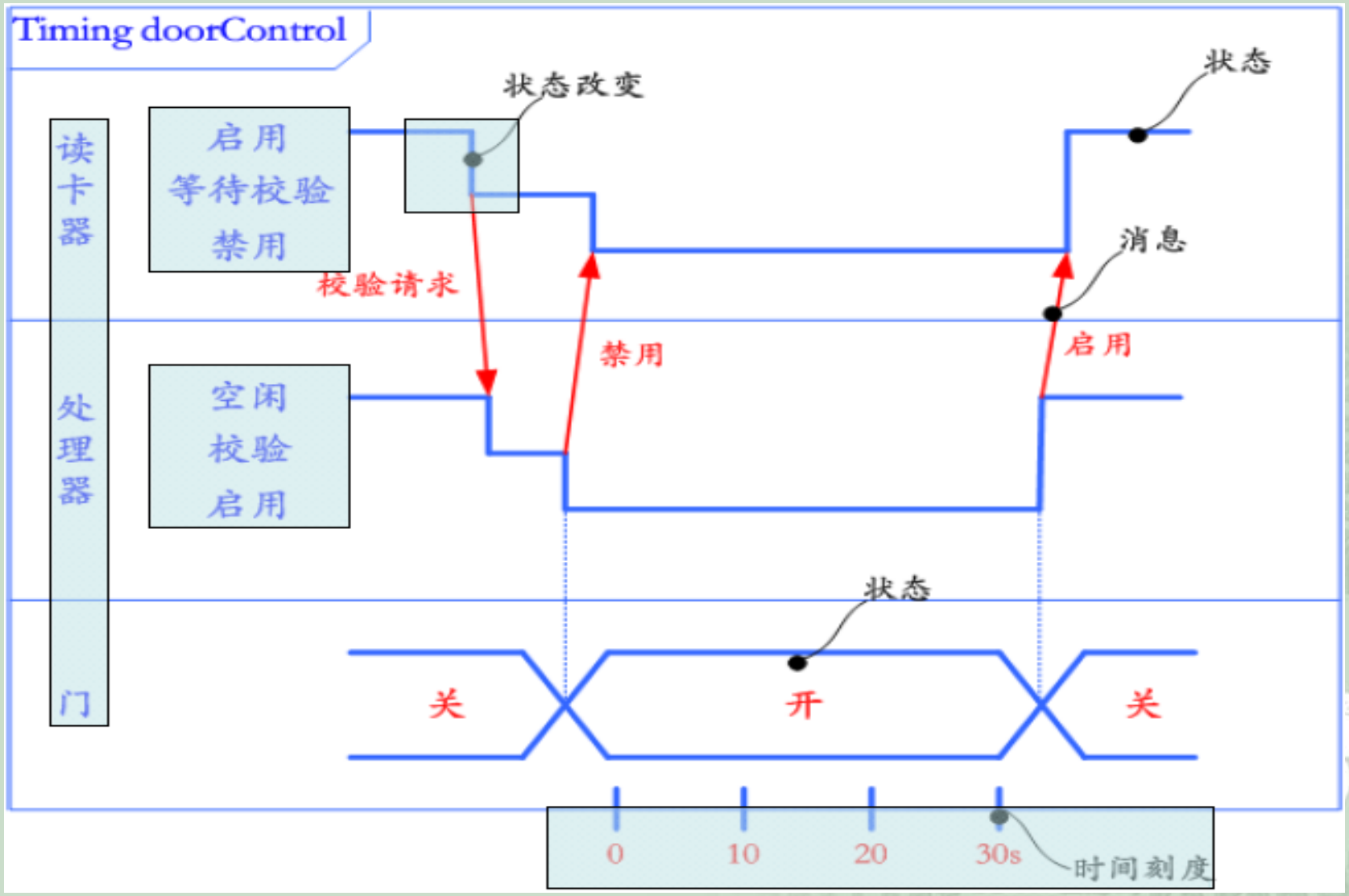
顺序图和通信图对比

- 顺序图强调消息的时间顺序，通信图强调参加交互的对象的组织，两者可以相互转换
- 顺序图不同于通信图的特征：
  - 顺序图有对象生命线
  - 顺序图有控制焦点
- 通信图不同于顺序图的特征：
  - 通信图有对象链接
  - 通信图必须有消息顺序号

Time diagram（定时图）

- 如果要表示的交互有很强的时间特性，使用定时图
- 定时图是一种特殊的顺序图，其与顺序图的区别在于：
  - 坐标轴交换了位置，改为从左到右来表示时间的推移
  - 用生命线的“凹下凸起”来表示状态的变化，每个水平位置代表一种不同的状态
  - 可显示一个度量时间值的标尺，用刻度表示时间间隔

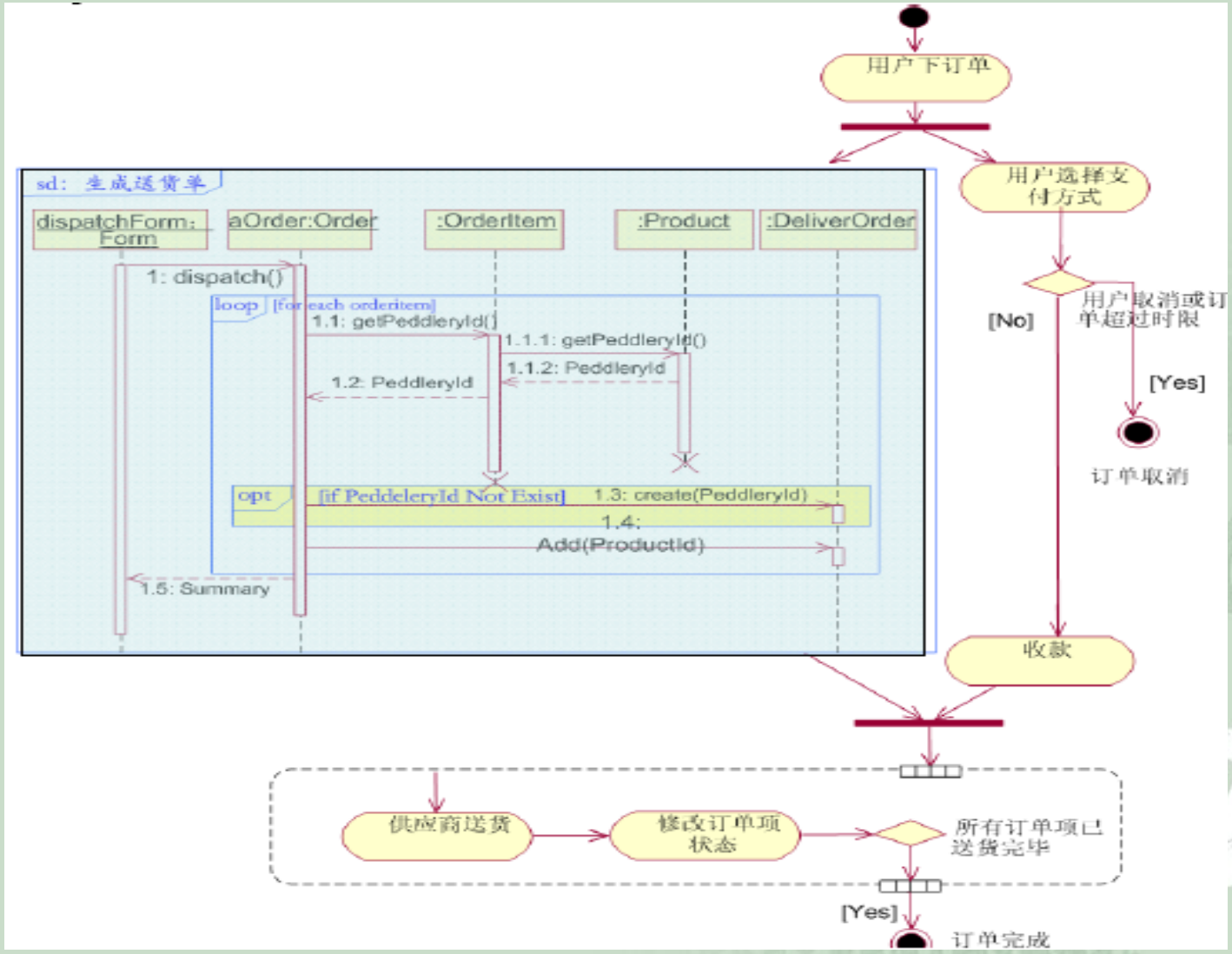
Time diagram示例



交互概述图

- 交互概述图是将活动图和顺序图嫁接在一起的图
- 可以看作活动图的变体
- 它将活动节点进行细化，用一些小的顺序图来表示活动节点内部的对象控制流
- 也可以看作顺序图的变体
- 它用活动图来补充顺序图

阅读交互概述图


用户下订单
用户选择支付方式
[No]
用户取消或订单超过时限
[Yes]
订单取消
sd: 生成送货单
dispatchForm: Form
aOrder:Order
:OrderItem
:Product
:DeliverOrder
1: dispatch()
loop [for each orderitem]
1.1: getPeddleryId()
1.1.1: getPeddleryId()
1.1.2: PeddleryId
1.2: PeddleryId
opt
[if PeddeleryId Not Exist]
1.3: create(PeddleryId)
1.4:
Add(ProductId)
1.5: Summary
收款
供应商送货
修改订单项状态
所有订单项已送货完毕
[Yes]
订单完成

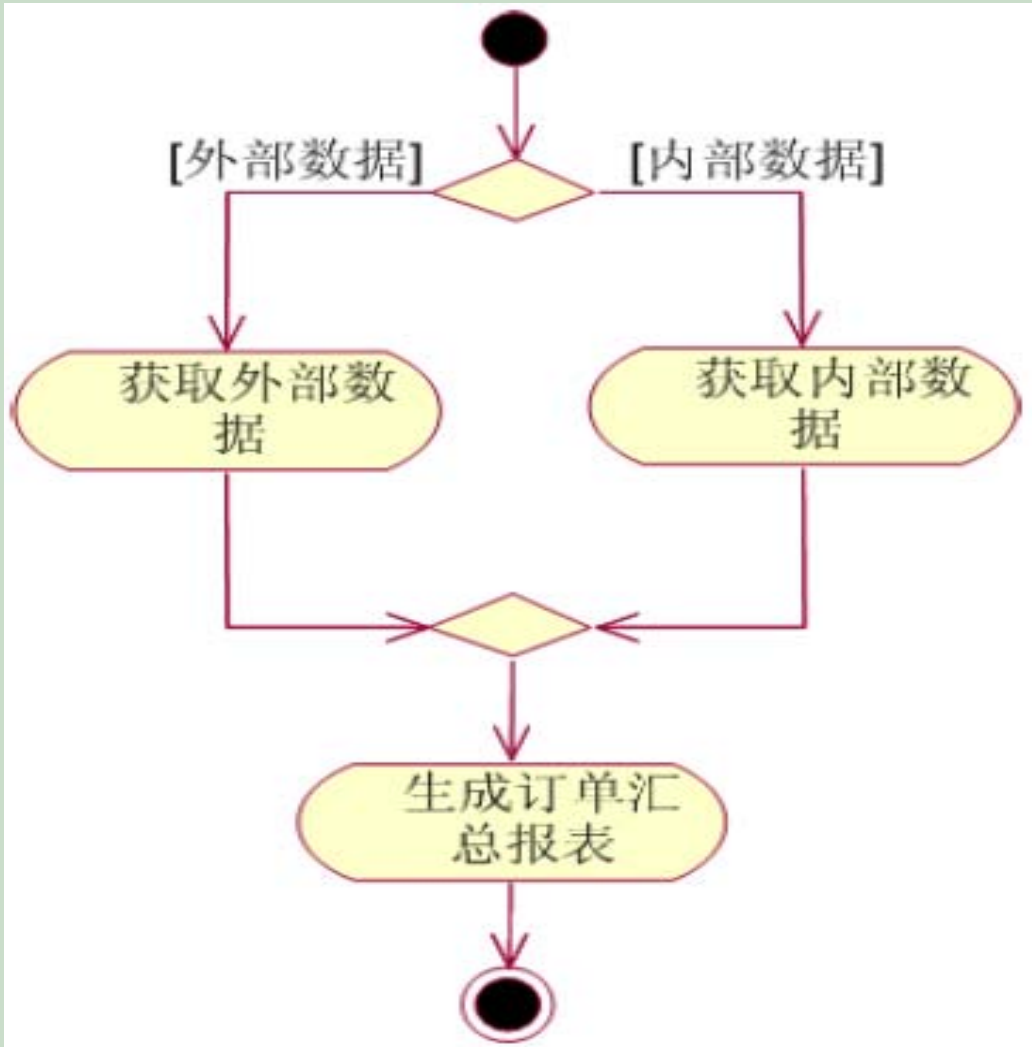
绘制交互概述图

- 决定绘制策略
- 用一种图理清主线
- 用另一种图来表述细节

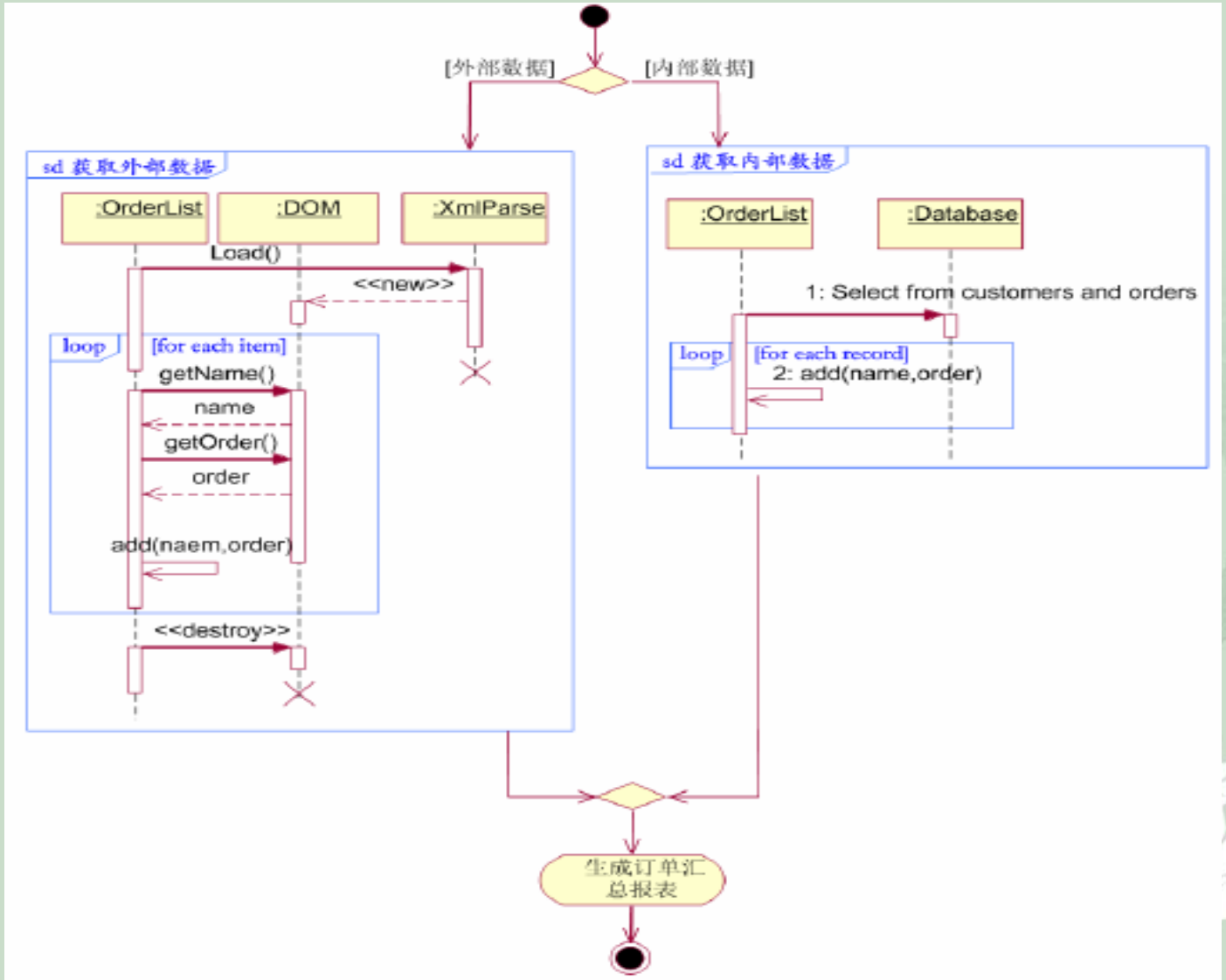
决定绘制策略

- 两种形式：一种是以活动图为主线，并用顺序图表述细节；另一种是以顺序图为主线，并用活动图来表述细节
- 如果是对工作流进行建模，那么应该先采用活动图来表示工作流的活动控制流，然后再通过顺序图来描述其中一些活动节点的对象控制流，阐述更多实现细节
- 如果是在为代码的设计、实现进行建模，那么可以先通过顺序图理清对象之间的控制流；然后再通过活动图来表示某些重要的方法、调用的算法流程

用活动图表述主线



用顺序图描述细节


[外部数据]
[内部数据]
sd 获取外部数据
:OrderList
:DOM
:XmlParse
Load()
<<new>>
loop
[for each item]
getName()
name
getOrder()
order
add(naem,order)
<<destroy>>
sd 获取内部数据
:OrderList
:Database
1: Select from customers and orders
loop
[for each record]
2: add(name,order)
生成订单汇
总报表

对象职责分配

- 交互图体现了如何为对象分配职责。当交互图创建时，实际上已经为对象分配了职责，反映到交互图就是该对象发送相应的消息到不同类的对象。
- 职责是在绘制交互图的时候考虑和决定的。类图的方法栏表示了职责分配的结果，职责具体由这些方法来实现。

对象职责分配

- GRASP：“通用职责分配软件模式”
- 其中5个:
  - Information Expert(信息专家)
  - Creator(创建者)
  - Low Coupling(低耦合)
  - High Cohesion(高内聚)
  - Controller(控制器)

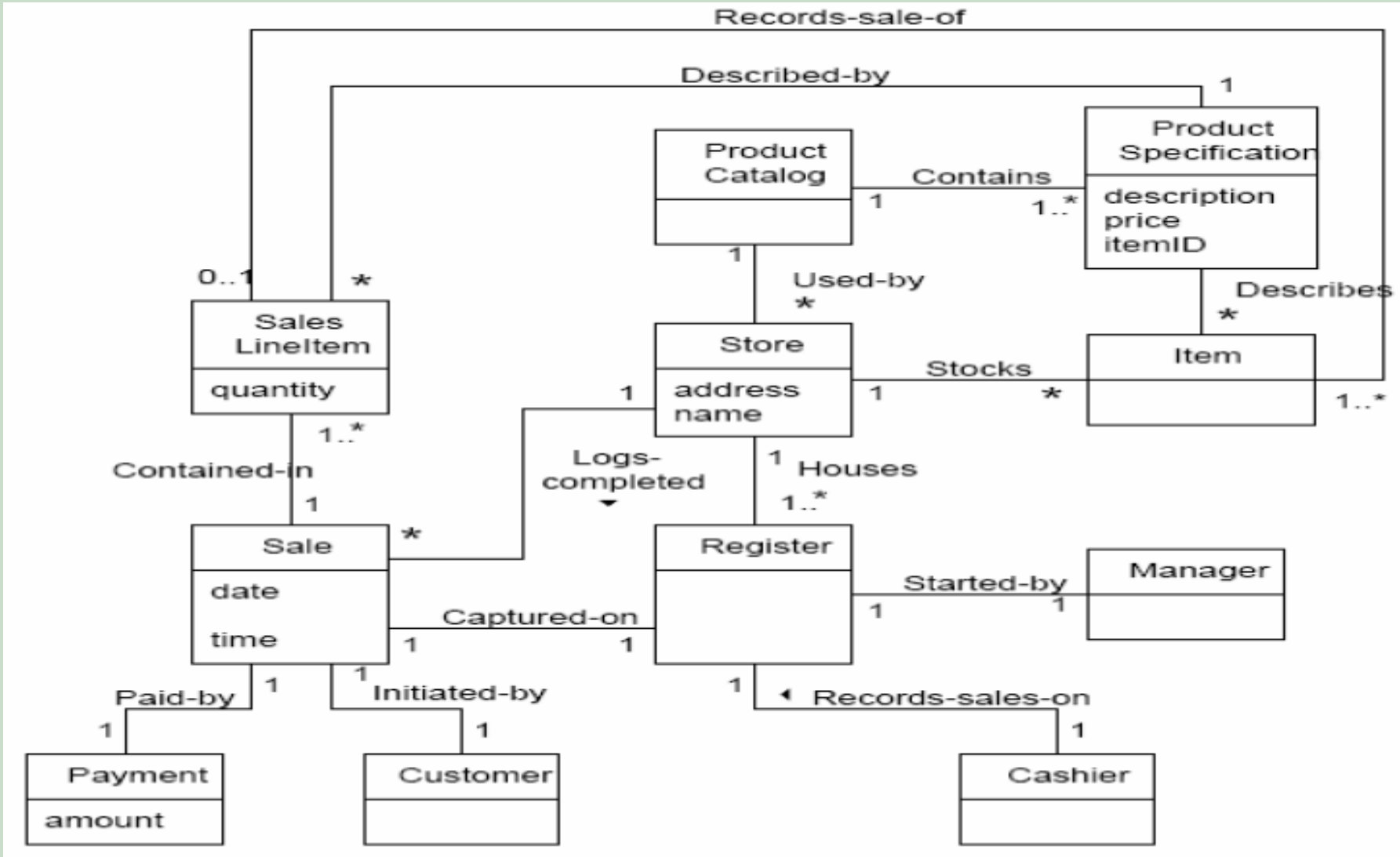
信息专家模式

- 信息专家模式核心思想：把职责分配给具有完成某项职责所需信息的个体。
- 信息专家模式在职责分配中使用得非常广泛，它是对象设计中经常使用的基本指导原则。
- 职责的履行需要信息，而信息往往分布在不同的对象中。这就意味着许多“部分”的信息专家有时需要协作才能来完成一个任务。

信息专家模式优点

- 因为对象使用自己的信息来实现服务，所以信息的封装性得以维持，也有利于实现低耦合/健壮性/易维护性。
- 行为分散在不同的类中，这些类各自具有完成行为所需要的信息。这些“轻量级”的类易于理解和维护。

举例：POS


Records-sale-of
Described-by
1
Product Specification
description
price
itemID
Product Catalog
Contains
1
1..*
1
Used-by
*
Describes
*
0..1
*
Sales LineItem
quantity
Store
address
name
Stocks
1
*
Item
1..*
1..*
Contained-in
1
Logs-completed
1
Houses
1..*
Sale
date
time
*
Register
Manager
Started-by
1
1
Captured-on
1
1
1
1
Paid-by
1
1
Initiated-by
1
1
Records-sales-on
1
Payment
amount
Customer
Cashier

举例：POS

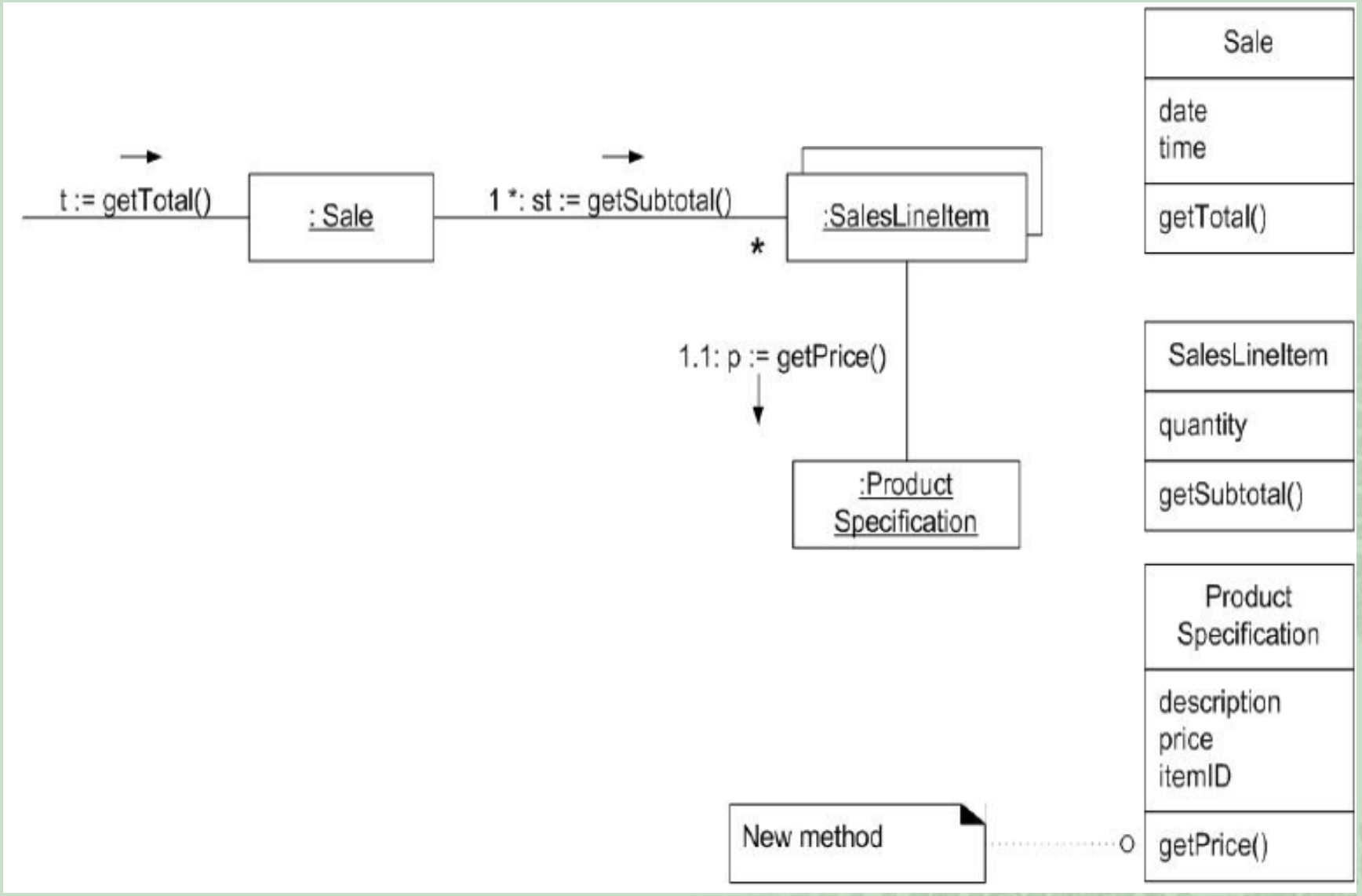
Q：应该有某个类知道Sale的total，那么这个职责应该分配给谁？

A：考虑计算Sale的total需要哪些信息？需要知道所有SalesLineItem及其它们的Subtotal。Sale包含SalesLineItem，因此Sale是信息专家。

考虑计算SalesLineItem的Subtotal需要哪些信息？SalesLineItem.quantity和ProductSpecification.price。

SalesLineItem知道其quantity和相关ProductSpecification，所以SalesLineItem是信息专家。……

举例：POS


t := getTotal()
: Sale
1 *: st := getSubtotal()
:SalesLineItem
*
1.1: p := getPrice()
:Product Specification
Sale
date
time
getTotal()
SalesLineItem
quantity
getSubtotal()
Product Specification
description
price
itemID
getPrice()
New method

创建者模式

- 创建者模式指导怎样分配“与创建对象”相关的职责。创建者模式的基本目的是找到一个在任何情况下都与被创建对象相关联的创建者，选择这样的类作为创建者能支持低耦合。
- 集合聚集了部分,容器包含了内容,记录器记录了被记录的数据，这些类之间的关系在类图中非常普遍。创建者模式建议：封装的容器类和记录器类是创建“自己包含或者记录的元素”的很好候选者。

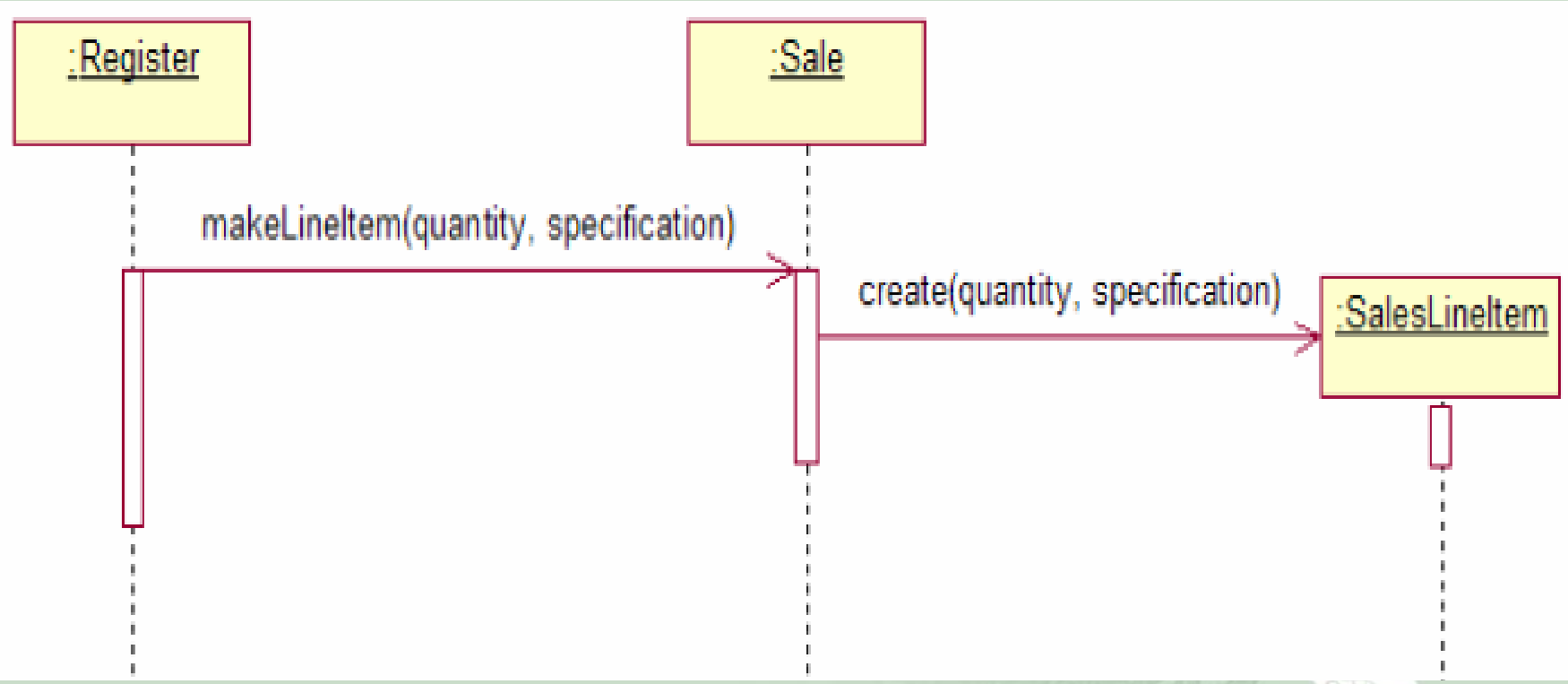
创建者模式优点

- 支持低耦合，这种设计意味着具有更低的维护依赖性和更高的重用机会。创建者模式之所以可能不增加耦合性是由于被创建类对于创建类而言已经可见了，正是因为已存在的关联使得它成为创建者。

举例：POS

- 哪个类应该负责产生SalesLineItem的实例？
- 根据创建者模式,我们应该寻找哪个类聚合或者包含了SalesLineItem类的实例。因为一个Sale类包含了许多SalesLineItem对象，所以创建者模式指明Sale类是创建SalesLineItem类实例的很好的候选者。

举例：POS


:Register
:Sale
makeLineItem(quantity, specification)
create(quantity, specification)
:SalesLineItem

低耦合模式

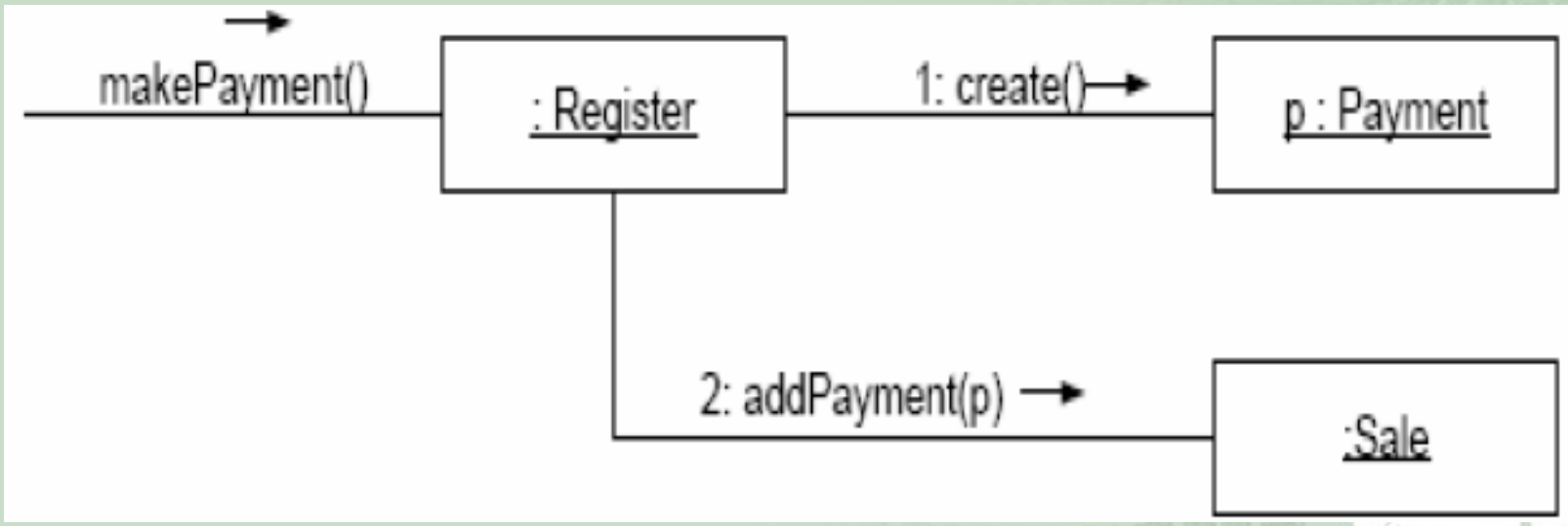
- 低耦合模式鼓励职责分配时不增加耦合性。从而避免了高耦合可能产生的不良后果。
- 优点：不受其它组件改变的影响；便于单独理解；重用方便

举例：POS

- 假设要创建一个Payment的实例并使它和Sale之间的实例关联，哪个类应当承担职责呢？在真实世界中Register记录了一个Payment的实例，根据创建者模式，Register类应该是创建Payment的候选类。

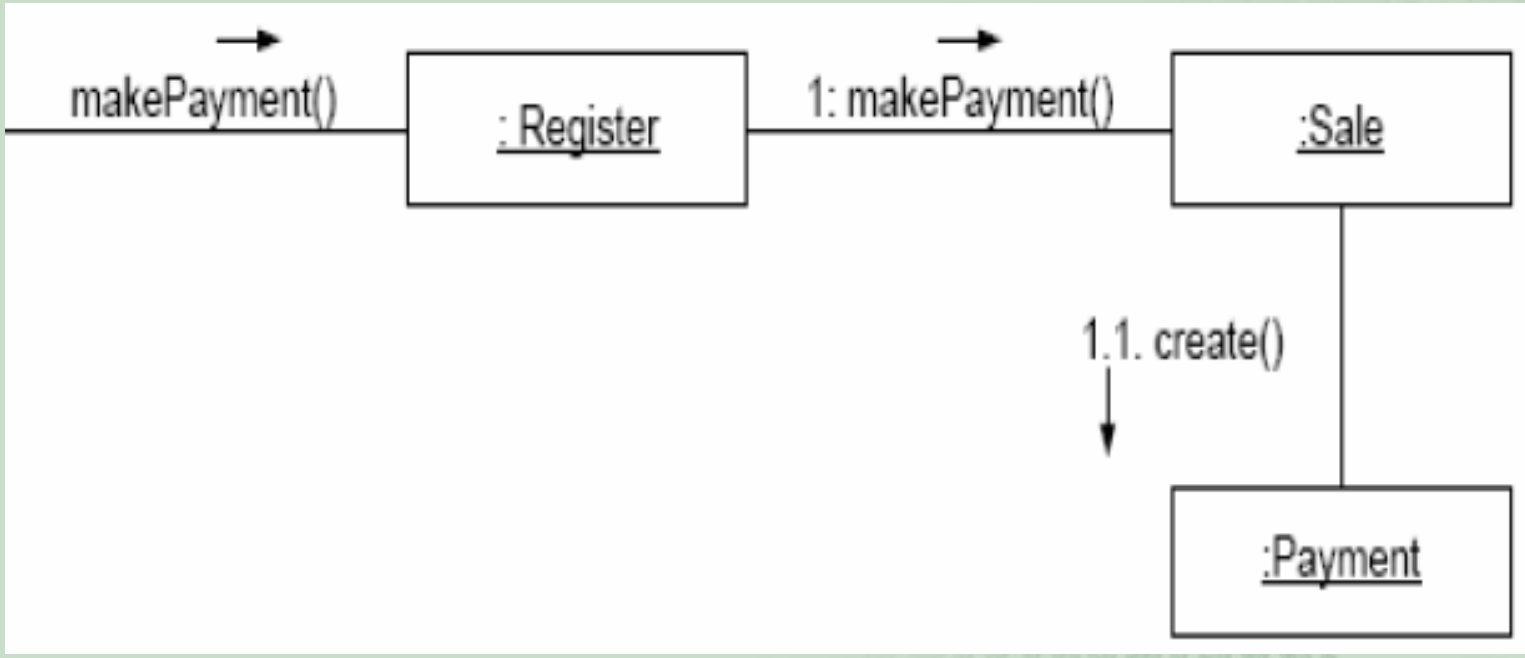
举例：POS

- 方案一：一个Register发送一个addPayment消息给Sale实例，并将新生成的Payment实例作为一个参数传递给Sale。Register需要知道Payment类的消息，两者被耦合。



举例：POS

- 方案二：一个Sale实例最终要和一个Payment实例耦合起来。因此Payment由Sale创建，没有增加耦合度，所以整体耦合度低。



高内聚模式

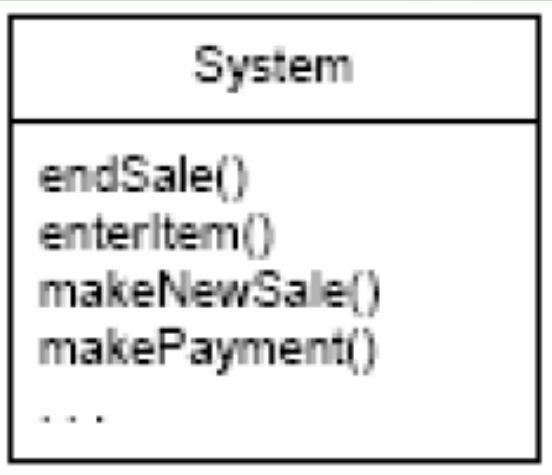
- 高内聚模式鼓励职责分配时保持较高的内聚性。
- 优点：使设计的清晰性和易于理解性得到提高；维护和扩展得到简化；常常支持低耦合；重用性提高。

控制器模式

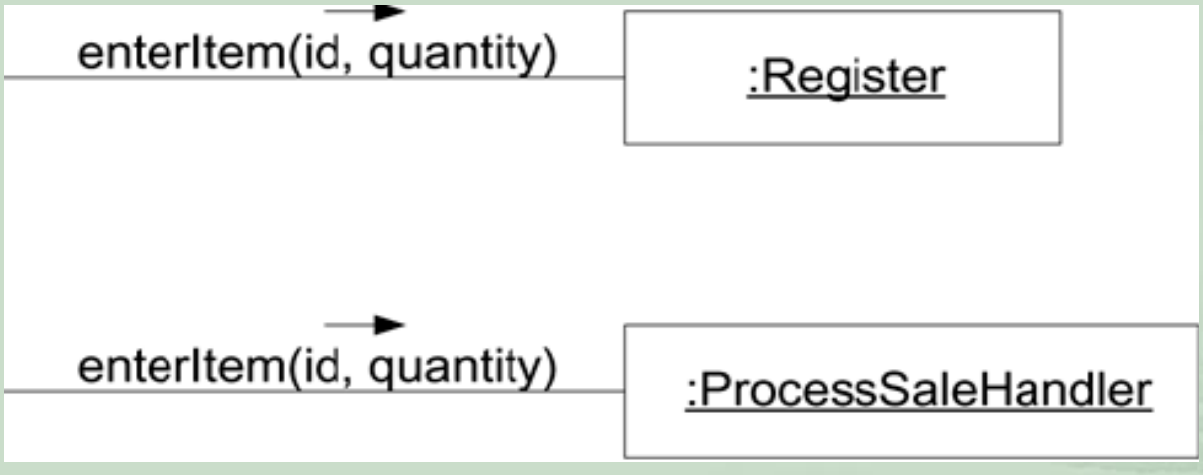
- 控制器是负责接收或者处理系统事件的对象。
- 把接收或处理系统事件的职责分配给：
  - 代表整个系统、设备或子系统的类
  - 表示用例场景的类，如< UCName>Handler，<UCName>Coordinator或<UCName>Session

举例：POS

- 在NestGen POS中，有几个系统操作如图所示。谁应该是enterItem和endSale这些系统事件的控制器？



举例：POS


enterItem(id, quantity)
:Register
enterItem(id, quantity)
:ProcessSaleHandler


Register
makeNewSale()
enterItem(itemID, qty)
makePayment(...)
endSale()

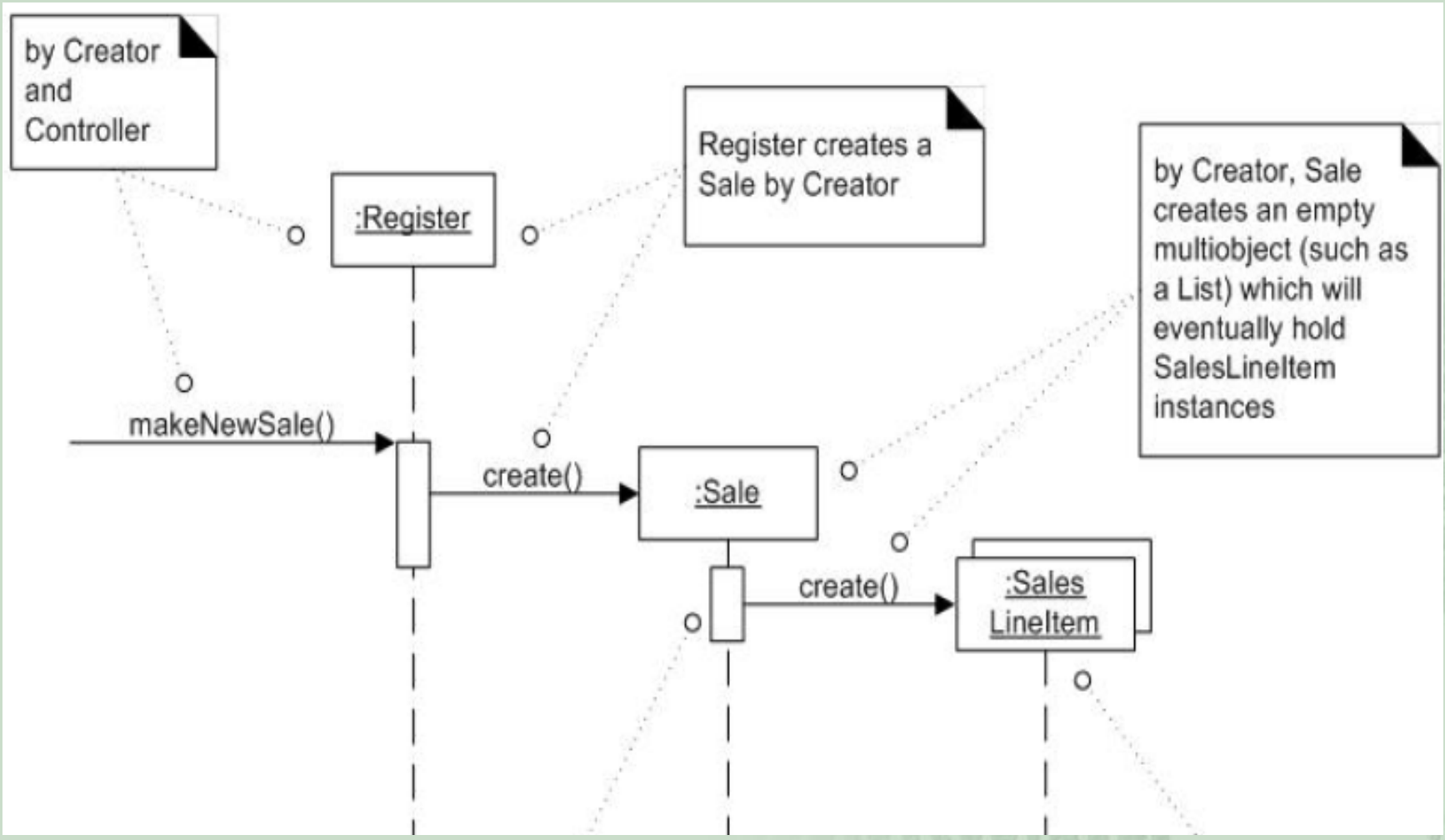
GRASP设计模式

- 信息专家—把职责分配给信息专家
- 控制者—把接收和处理系统事件消息的职责分配给控制者
- 创建者—把创建类A实例的职责分配给一个在任何情况下都与被创建对象相关联的创建者
- 低耦合—分配职责时要保持低耦合
- 高内聚—分配职责时要保持高内聚

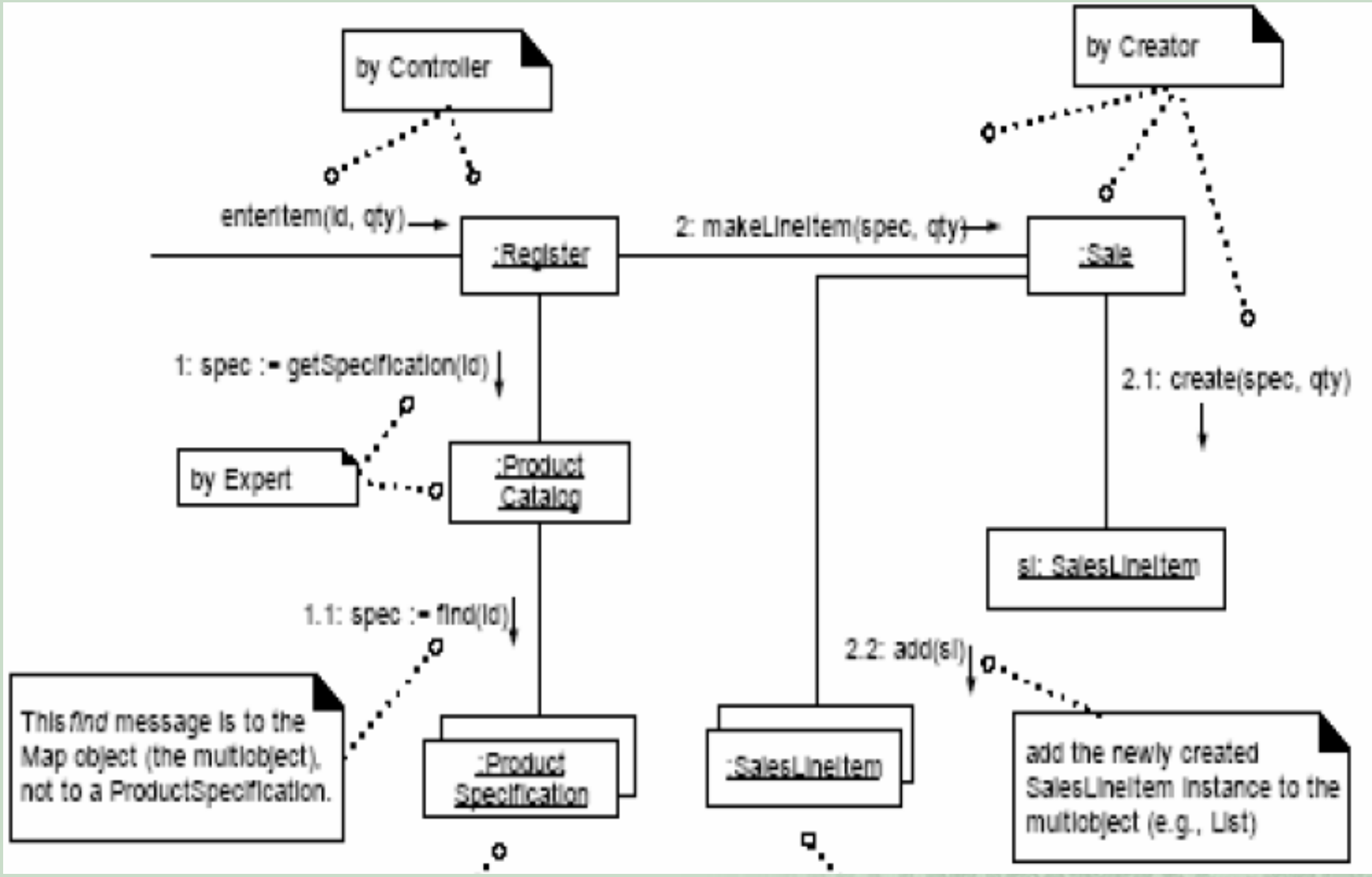
GRASP的应用

- 除非是控制器或创建者问题，信息专家模式是应该第一考虑采用的模式， 再利用低耦合和高内聚进行优化设计。

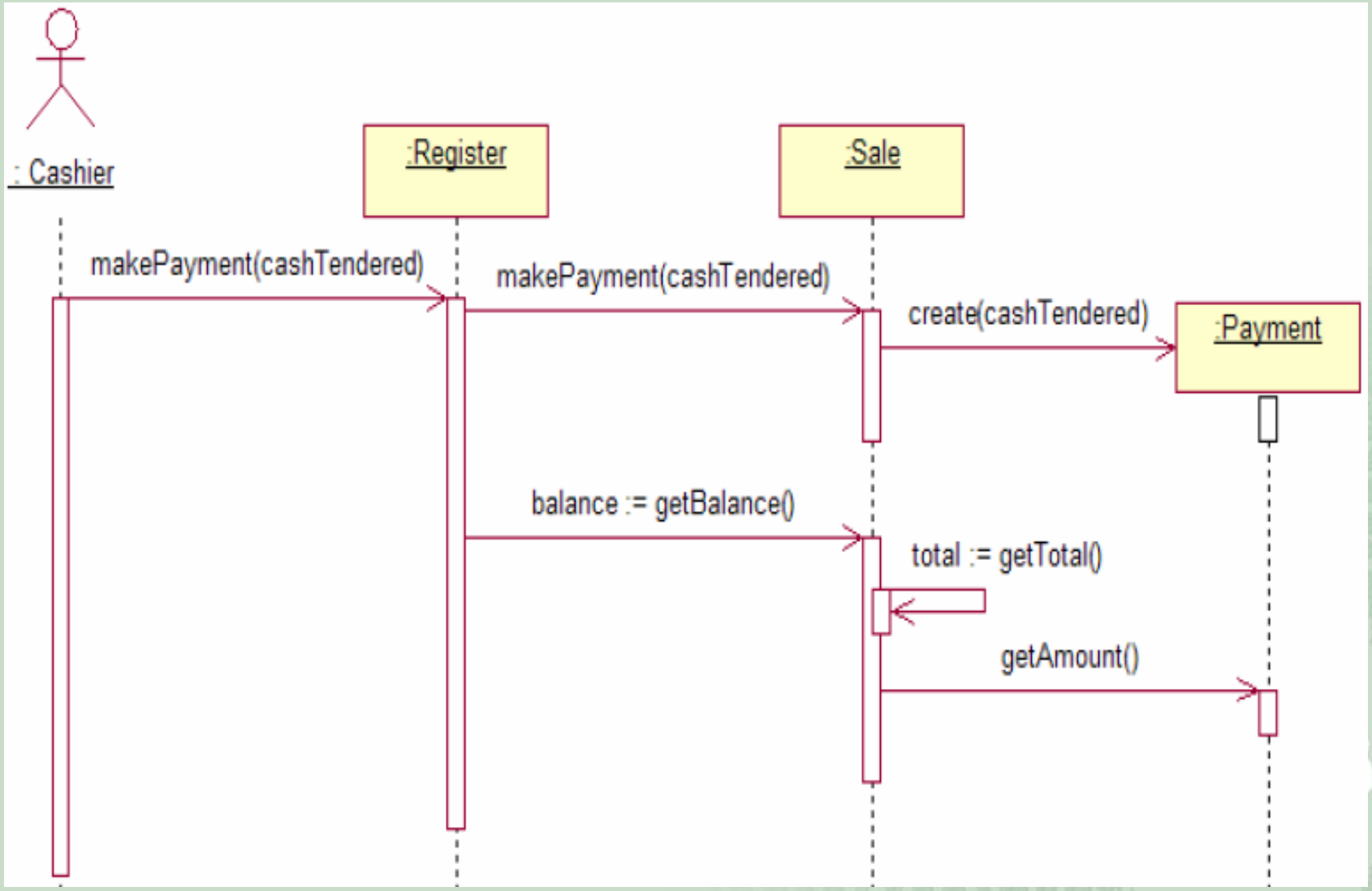
举例：POS


by Creator and Controller
:Register
Register creates a Sale by Creator
by Creator, Sale creates an empty multiobject (such as a List) which will eventually hold SalesLineItem instances
makeNewSale()
create()
:Sale
create()
:Sales LineItem

举例：POS


by Controller
by Creator
enterItem(id, qty)
:Register
2: makeLineItem(spec, qty)
:Sale
1: spec := getSpecification(id)
2.1: create(spec, qty)
by Expert
:Product Catalog
sl: SalesLineItem
1.1: spec := find(id)
2.2: add(sl)
This find message is to the Map object (the multiobject), not to a ProductSpecification.
:Product Specification
:SalesLineItem
add the newly created SalesLineItem instance to the multiobject (e.g., List)

举例：POS


: Cashier
:Register
:Sale
:Payment
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
balance := getBalance()
total := getTotal()
getAmount()