**1.OOA、OOD、OOP:**

**（1）OOA：object-oriented analysis，面向对象分析，在这个阶段，开发者是被和定义系统中的对象、他们的关系以及他们的行为。常用UML用例图和类图。**

**（2）OOD（object-oriented design），面向对象设计，基于OOA的结果，开发者创建更详细的UML类图、序列图和状态机图来设计系统的结构和行为。**

**（3）OOP（object-oriented programming），面向对象编程，在该阶段，开发者使用编程语言（如java、C++等）实现OOD阶段定义的设计。**

**2.RUP（rational unified process）统一软件开发过程，是一个迭代软件开发过程框架。核心特点是：迭代开发、用例驱动、基于架构、风险驱动。包含四个阶段：inception、elaboration、construction、transition**

**3.UML（unified modeling language）统一建模语言：描述、构造和文档化系统制品的可视化语言。UML是图形化表示法的事实标准，用来绘制和展示与软件相关的图形（以及文字）。**

**4.LSP（Liskov substitution principle，利斯科夫可替代原理）：**

**（1）内容：在软件中，如果S是T的子类，那么程序中的对象T应该可以被替换成对象S，而不会影响程序的正确性、行为或功能。**

**即LSP要求子类在替换基类时不会引入错误或不兼容。要求：**

**①行为一致性：子类保持基类的行为**

**②输入参数和返回值类型一致性**

**③前置条件和后置条件：子类的前置条件不能比基类的更严苛；子类的后置条件不能少于基类的后置条件。**

**④例外处理一致性**

**（2）LSP有助于提高代码的可维护性和可扩展性。**

**5.设计模式：**

**（1）定义：一组用于解决与软件设计中常见问题的通用可重用解决方案。常见的设计模式有GRASP模式和GoF模式。而GoF模式通常分为三类：创建型模式、结构性模式、行为性模式；GRASP模式有9种：低耦合模式、高内聚模式、控制器模式、多态模式、防止变异模式、纯虚构模式、创建者模式、信息专家模式、间接性模式**

**第三部分 细化迭代1——基础**

【小结】细化阶段的制品（不包含初始阶段开始构建的制品）：

①领域模型——领域概念的可视化~~**没有方法的类图，不是软件类**

②设计模型——描述逻辑设计的一组图，包括软件类图、对象交互图、包图

③数据模型、软件架构文档……

**第九章 领域模型（domain model）**

目标：确定概念类->创建初始的领域模型->建立属性和关联

【小结】**领域模型是图形——使用没有操作的UML类图**来表示，不是文本。

在每个迭代中，领域模型都会先定于之前和当前要考虑的场景。领域模型通常开始和完成于细化阶段。

**一、一些基础理解**

1.定义：领域模型（也称为概念模型、领域对象模型、分析对象模型）是对领域内的概念类或现实世界中对象的可视化表示。

【注】**领域模型指的是对现实世界概念类的表示，而非软件对象的表示**。

2.领域模型提供了概念透视图，展示了：领域对象或概念类、概念类之间的关联、概念类的属性。

（3）概念类（conceptual class）

通俗地讲是思想、事物或对象。正式的讲可以从其符号、内涵和外延来考虑：

符号：表示概念类的词语或图形

内涵：概念类的定义

外延：概念类所适用的一组示例

3.为什么要创建领域模型？

降低与OO建模之间的表示差异~OO的关键思想：领域层软件类的名称要源于领域模型中的名称。

**二、如何创建领域模型**

（一）找概念类

法一：重用和修改**现有的模型**

法二：使用**分类列表**

通过制作概念类候选列表来开始创建领域模型，并在分析时建立一些优先级。

法三：通过识别**名词短语**寻找概念类

1.**语言分析（linguistic analysis）**——在对领域的文本性描述中识别名词和名词短语，将其作为候选的概念类或属性。

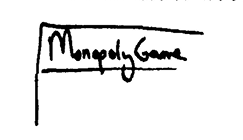
~~语言分析适用于详述形式用例中的描述

2.注：不可能存在名词到类的映射机制，并且自然语言中的词语具有二义性——不同名词短语可能表示统一概念类或属性，或者具有歧义。

3. 【注】如果我们认为某概念类X**不是现实世界中的数字或文本**，那么X可能是**概念类**而不是属性。

e.g.商店在现实世界不会被认为成数字或文本，所以Store应该是概念类

目的地机场在现实世界不会被认为成数字或文本，所以Airport应该是概念类

（二）描绘概念类

1.使用敏捷建模的思想——绘制类图的草图（让类框的底部和右侧开放状态，如右图）

2.报表对象——模型中是否要包括“票据”？

（1）排除：在领域模型中显示其他信息的报表并没有意义，因为其所有信息都是源于或复制于其他信息源，e.g.收银员何时查看了某件商品的价格。

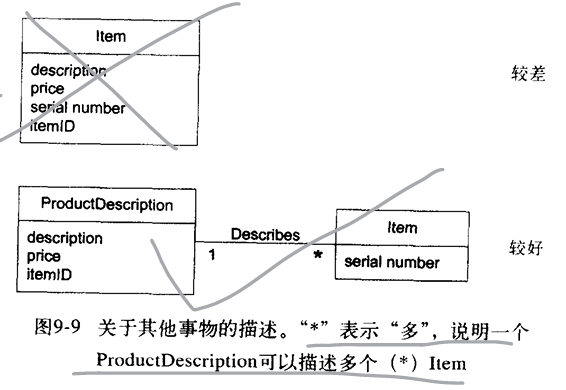
（2）保留：通常持有票据的人有退货的权利。

3.使用领域术语

**（三）描述类description class**

1.描述类包含描述其他事物的信息~~项目-描述符（item-descriptor）模式

2.为什么需要描述类？

假设Item表示商店里的实际商品，拥有一个序列号、描述、价格和ID，且假设这些内容不会在任何其他地方记录。同是假设每售出一件实际的商品，响应的Item软件实例就会从“软件领域”删除。比如汉堡店中汉堡全卖完了，那么对应的所有Item实例都会被删除，那如果此时有人问汉堡多少钱，会因为没有Item实例而无法回答汉堡多少钱。

3.故对于上述问题，我们需要ProductionDescription类来记录商品的信息。ProductionDescription并不代表Item，而是表示有关商品的描述信息。

4.何时需要描述类？

（1）需要有关商品或服务的描述，独立于任何商品或服务的现有实例

**（2）删除其所描述事物的实例后，导致信息丢失**

（3）减少冗余或重复的信息

**——————————三、关联association———————**

（一）相关概念

1.定义：关联是类（更准确的说，是类的实例）之间的关系，表示有意义和值得关注的连接。在UML中，关联被定义为：两个或多个类元之间的语义联系，涉及这些类元实例之间的连接

2.由于领域模型是从概念角度出发的，所以**是否需要记录关联，要基于现实世界的需要**，**而不是基于软件的需要**。

~~如果存在**需要保持一段时间的关系** ，则将这种语义表示为关联

3.避免加入大量的关联

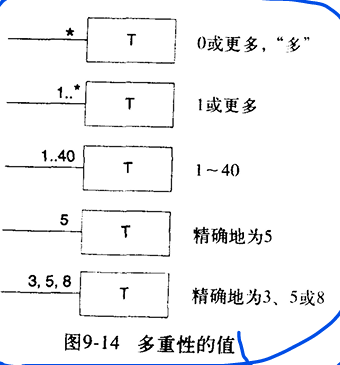
4.【注】（1）在领域建模过程中，关联**不是**关于数据流、数据库外键联系、实例变量或**软件方案中的对象**连接的语句；**关联**声明的是**针对现实领域**从纯概念角度看有意义的关系。

（2）领域模型不是数据模型，添加关联是为了突出我们对重要关系的大致理解，而**非**记录对象或数据的结构。

（二）关联表示法

1.关联表示为类之间的连线，并冠以**首字母大写的关联名称**：（一般左->右，从上->下读）

~~**关联命名：“类名-动词短语-类名”，动词短语：e.g.has、use。**

****2.关联的末端可以包含多重性表达式，用于指明类的实例之间的数量关系。

~~**多重性（multiplicity）①定义了类A有多少实例可以和类B的一个实例关联。**

②多重性的值表示在**特定时刻**（**而不是在某个时间跨度内**）有效关联的实例数量（多重性的值见右图）

③多重性的值和建模者和软件开发者的关注角度有关，因为它表达了将要（或可能）在软件中反应的领域约束，即多重性和语境有关。

3.关联本质上是双向的，其每一端称为角色（具有可选项：多重性表达式、名称、导航）。

~~两个类之间可能有多重关联。

**4.常见关联列表：**

A是与交易B相关的交易；e.g.现金支付­—销售

A是交易B中的一个项目；

A是交易（或项目）B的产品或服务；

A是与交易B相关的角色；e.g.顾客—支付、收银员—票据

A是B的物理或逻辑部分；e.g.方格—棋盘；座位—飞机

A被逻辑或物理地包含在B中；方格—棋盘；寄存器—商店

A是B的描述；产品描述—商品

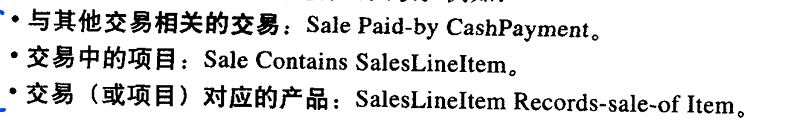
A在B中被感知/记日志/记录/生成报表/捕获；e.g销售—寄存器

A是B的成员；e.g.收银员—商店；玩家—游戏

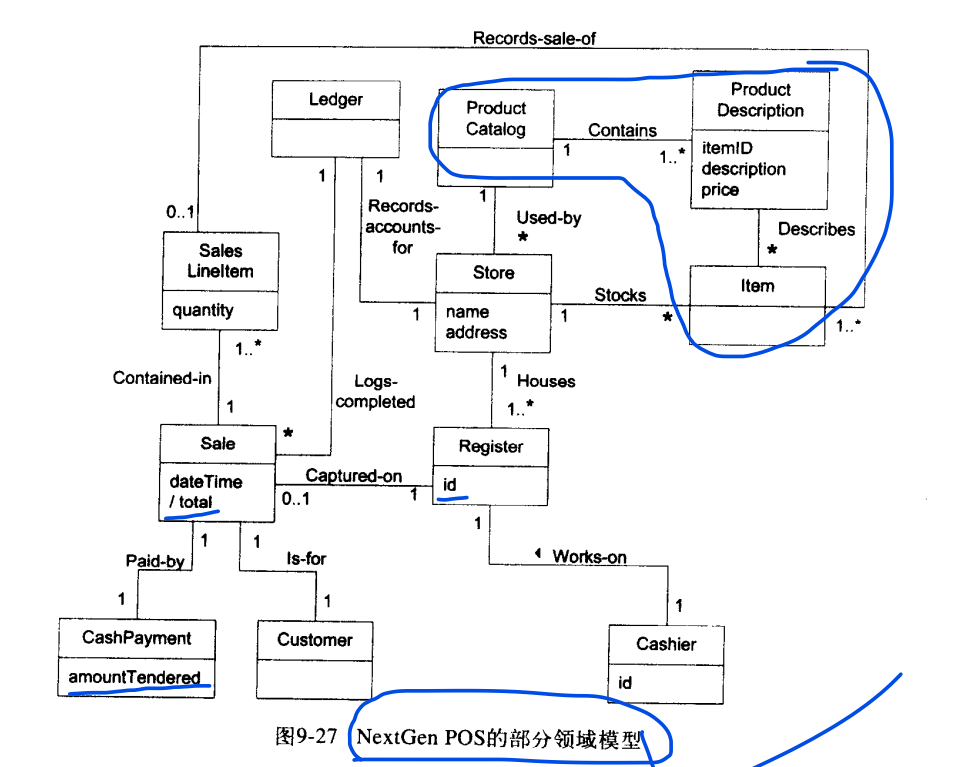
A是B的组织化子单元；e.g.分店—商店

A使用/管理/拥有B；e.g.收银员—寄存器；飞行员—飞机

A与B相邻；e.g.方格—方格；城市—城市

5.例：领域模型中的关联

下图是NectGen POS中的部分领域模型：

**—————————四、属性attribution————————**

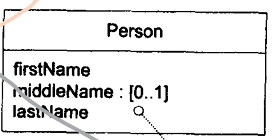
（一）定义：属性是对象的逻辑数据值，能够满足当前所开发场景的信息需求。

e.g.在处理销售用例中的票据通常含有日期和时间、店名和地址以及收银员ID等：

Sale需要dateTime属性；Store需要name和address属性；Cashier需要ID属性。

（二）表示

1.属性的完整语法是：

**可见性 名字：类型 多重性=默认值 {特性表}**

①可见性：“+”表示公有；“-”表示私有；

②多重性：通过表达式表示可选值，如右图中[0..1]指该属性可以出现1个值或0个值，其意义是说一个人可以没有中名

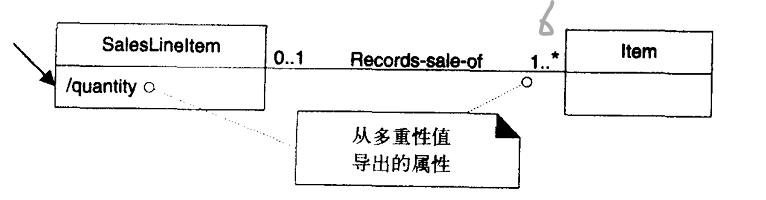
③默认值指的是初始值，是指对象在初始化时赋予属性的值

④特性表一般是readOnly

2.**导出属性derived property**

（1）导出属性指定是可以由其他信息导出的属性。

（2）表示：在属性名称前加一个“/”

e.g.收银员收到6包豆腐，那么可以输入一次itemID，然后输入数量6->一个SaleLineItem可以和商品的多个实例关联。收银员输入的数量可以被记录为SaleLineItem的属性，也可以从关联的多重性的实际值计算出来（如下图）

**（三）数据类型属性与数据类型类**

1.大部分属性类型应该是“简单”数据类型，即属性的类型不应该是复杂的领域概念。

~~**数据类型（data type）：**

①定义：数据类型指的是一组值，这组值的表示本身不具有任何含义（在我们的模型或系统的语境下）。

②**对数据类型的等价性检验基于值**，而非标识，即如果来年各个数据类型的值相同，那就认为这两个实例是一个个体。

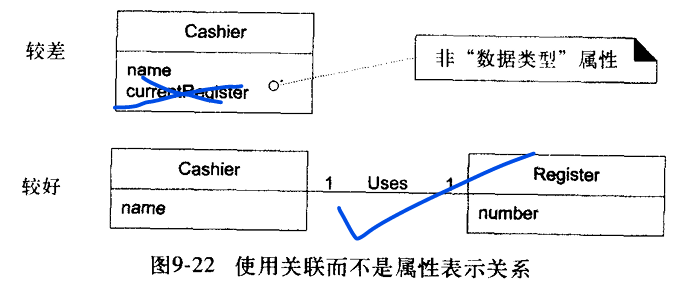
~在Java中，对值的等价性检验用equals方法，对标识符的等价性检验通过“==”。

③e.g.Boolean、Date（或DateTime）、Number、Character、String(Text)、Time；Address、Color、Geometrics(Point、Rectangle)、Phone Number、Social Security Number、Universal Product Code(UPC)、SKU、ZIP或邮政编码、枚举类型。

【注】在领域模型中建议类型主要为数据类型，并不意味着在编程语言中的属性只能是简单的基本数据类型。（因为领域模型是概念透视图，不是软件透视图。）在设计模型中，属性可以是任何类型。

2.应通过关联而不是属性来表示概念类之间的关系。

e.g.（见下图）Cashier类中的currentRegister属性是不合适的，因为currentRegister的类型是Register，并不是简单数据类型（Number或String）。所以应该使用关联而不是属性。

3.何时定义新的**数据类型类**？

**【注】数据类型类不一定被单独画成一个类。**

**（1）下述情况中，把最初认为是数字或字符串的数据类型表示为新的数据类型类：**

①**由不同的小节组成**，e.g.电话号码、人名；

②具**有与之相关的操作**，如解析或校验，e.g.社会安全号；

③具**有其他属性**，e.g.促销价格（可能有开始日期和结束日期）；

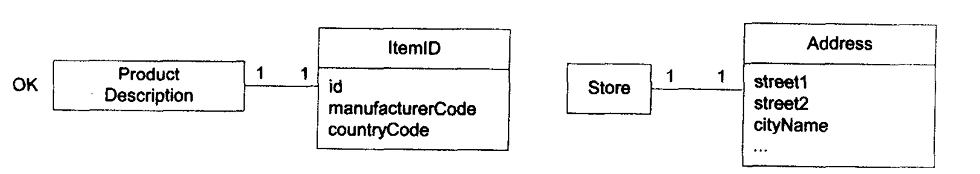
④**单位的数量**，e.g.支付总额具有货币单位；price和amount属性的数据类型应该是Money类（∵它们是货币单位的数量）

⑤具有以上性质的一个或多个类型的抽象，e.g.销售领域的商品标识符是诸如UPS和EAN这样的类型的泛化。

e.g. NectGen POS系统需要itemID属性，可以作为Item或ProductDescription的属性。注：itemID看起来只是数字或字符串，但实际上，商品项目标识符具有子域，应该在领域模型中加入类itemID（或者itemIdentifier），并将这个类作为itemID属性的类型。

（2）到底是在类框中表示数据类型类还是单独画出这个类再与其他类相关联？

①e.g.ItemID是数据类型,当它的实例的唯一标识不用于等价性检验时，就可以旨在类框的属性部分中提到这个数据类型类：

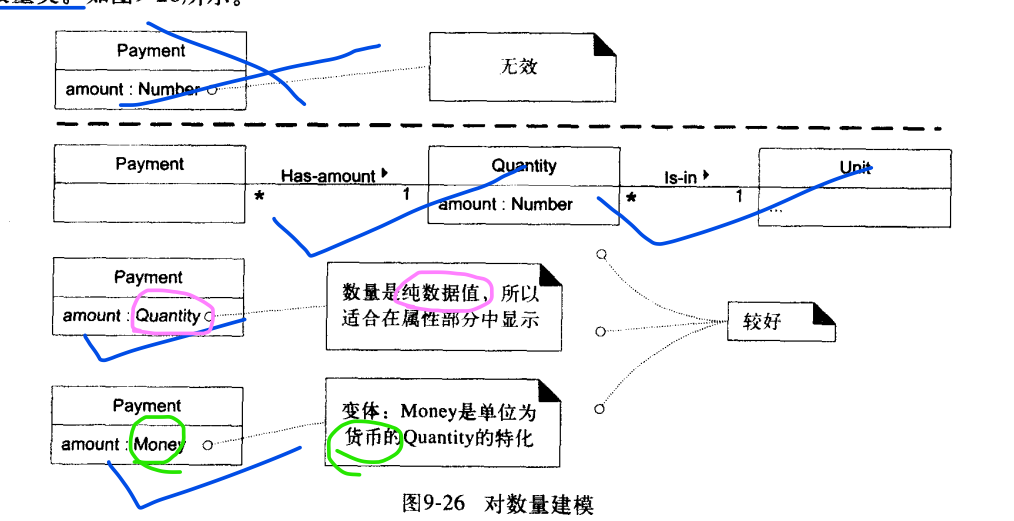
②如果ItemID具有属性和关联，那就需要将它表示为单独的概念类：

4.任何属性不表示外键——就是说一个类的属性不应该直接出现在其他类的属性框中，而是应该通过关联表示。

~~外键属性（foreign key attribute）——不能使用

5.对数量和单位建模

（引言）大部分用数字表示的数量不应该表示为纯数字，而应该给这些数量加上单位 ，并且通常还需要知道单位之间的转换关系。

一般，可以把纯数据值表示为Quantity类，并关联到Unit类；Money是一种单位为货币的数量类；Weight是单位为kg或磅的数量类（如下图）

【说明】其中的Money和Quatity和Weight类可以不用单拎出来说明。

**第十章 系统顺序图（SSD）**

目标：确定系统事件->为用例场景创建系统顺序图

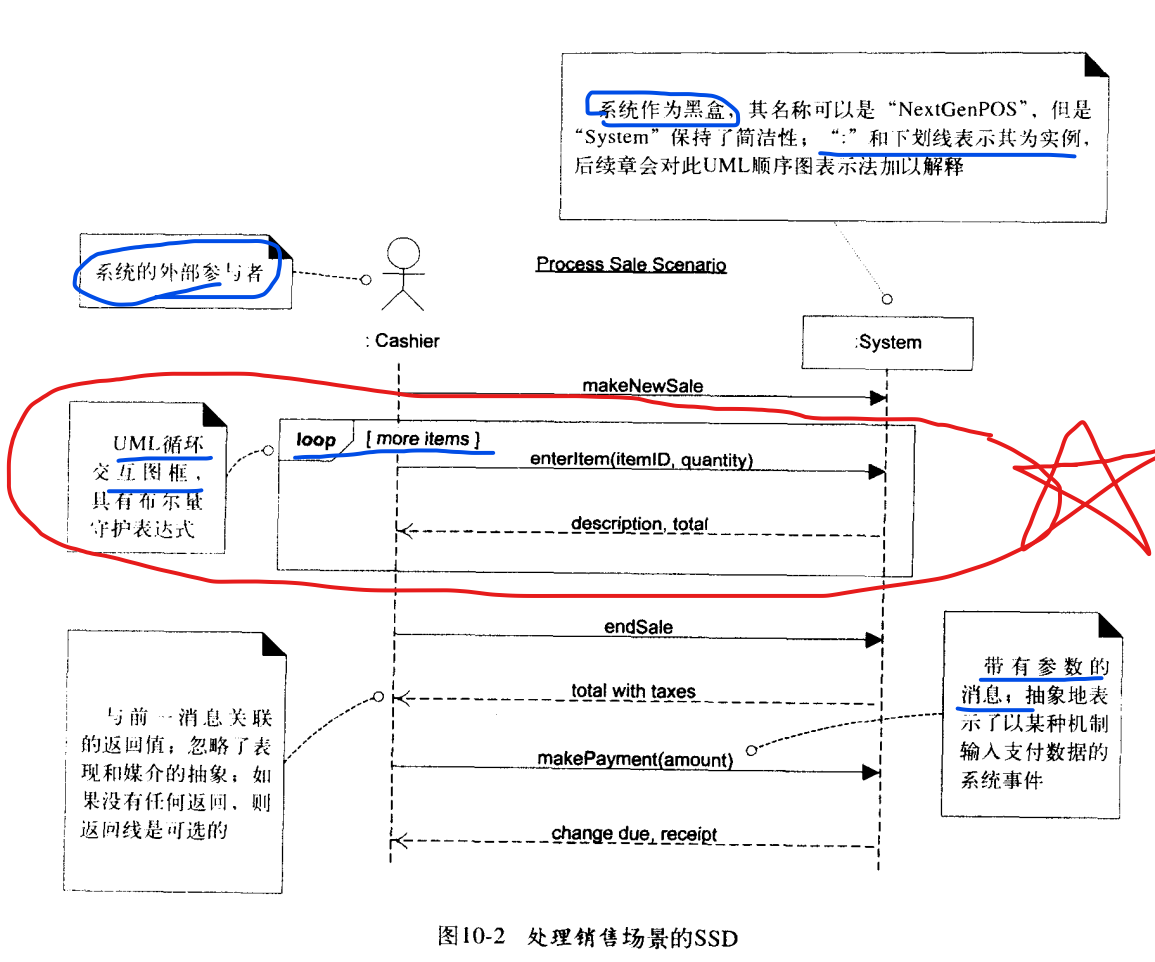
**【小结】SSD由用例导出，表示了一个场景**，但不用为所有场景创建SSD。

1.系统顺序图表示的是，对于用例的一个特定场景，直接与系统交互的外部参与者发生的事件，其顺序和系统之内的事件。所有系统被视为黑盒，SSD强调的是从参与者到系统的跨越系统边界的事件。（时间顺序自上而下）

~~系统行为（system behavior）：描述系统做什么，而无需解释如何做。

2.SSD中的操作可以在操作契约中进行分析，在词汇表中详细描述，并作为设计协作对象的起点。

~~对于大多数制品来说，一般在词汇表中描述其细节。

3.**交互框（interaction frame）**——用于表示顺序图中的循环

4.系统事件（system event）

（1）定义：外部输入的事件

（2）系统事件应该在意图的抽象级别而非物理的输入设备级别来表达

（3）系统事件的命名：以动词开始，e.g.增加、输入、结束、产生，以此强调这些事件是命令或请求。

**第十一章 操作契约operation contract**

**【小结】①如何编写契约：从SSD中确定系统操作->将复杂的结果不明显的或者在用例中不清楚的系统操作作为编写对象->描述后置条件（过去时）：创建和删除实例、修改属性、形成和清除关联。**

②SSD（系统顺序图）中的是系统事件，而在操作契约中描述的是系统操作。操作契约中的系统操作对应于SSD中的系统事件

③虽然用例文本中也有后置条件，但是用例文本只是对后置条件做了简述，而**操作契约详细描述后置条件**。

④操作契约是在细化阶段开始编写的。

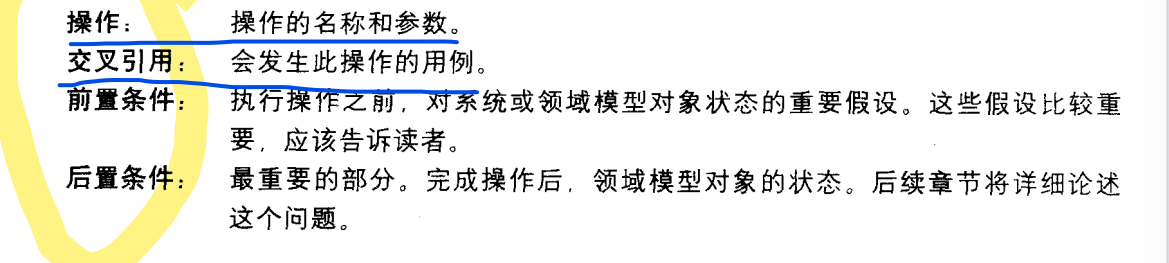
1.操作契约：对用例指出的**系统操作**的效用提供了更详细的分析。

（1）系统操作：

①系统操作是作为黑盒构件的系统在其公共接口中提供的操作。

②系统操作可以在绘制SSD时确定（∵SSD中展示的输入的系统事件对应处理该事件的系统操作）

（2）契约能够详细描述系统操作所需的变化，而无需描述做这些操作是如何完成的。

2.操作契约包括：

**———————————（1）后置条件（postcondition）—————————**

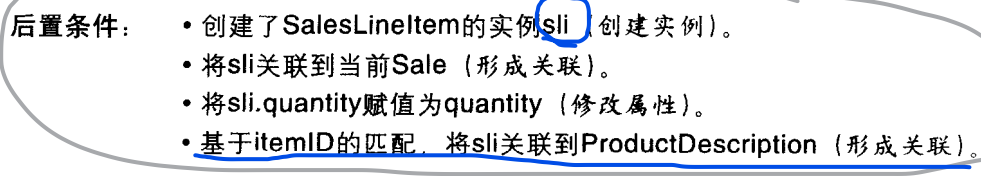
①定义：后置条件描述了领域模型内对象状态的宝海。领域模型状态变化包括创建实例、形成或消除关联以及改变属性。

【注】不要忘记在已有或新创建的对象之间建立关联。

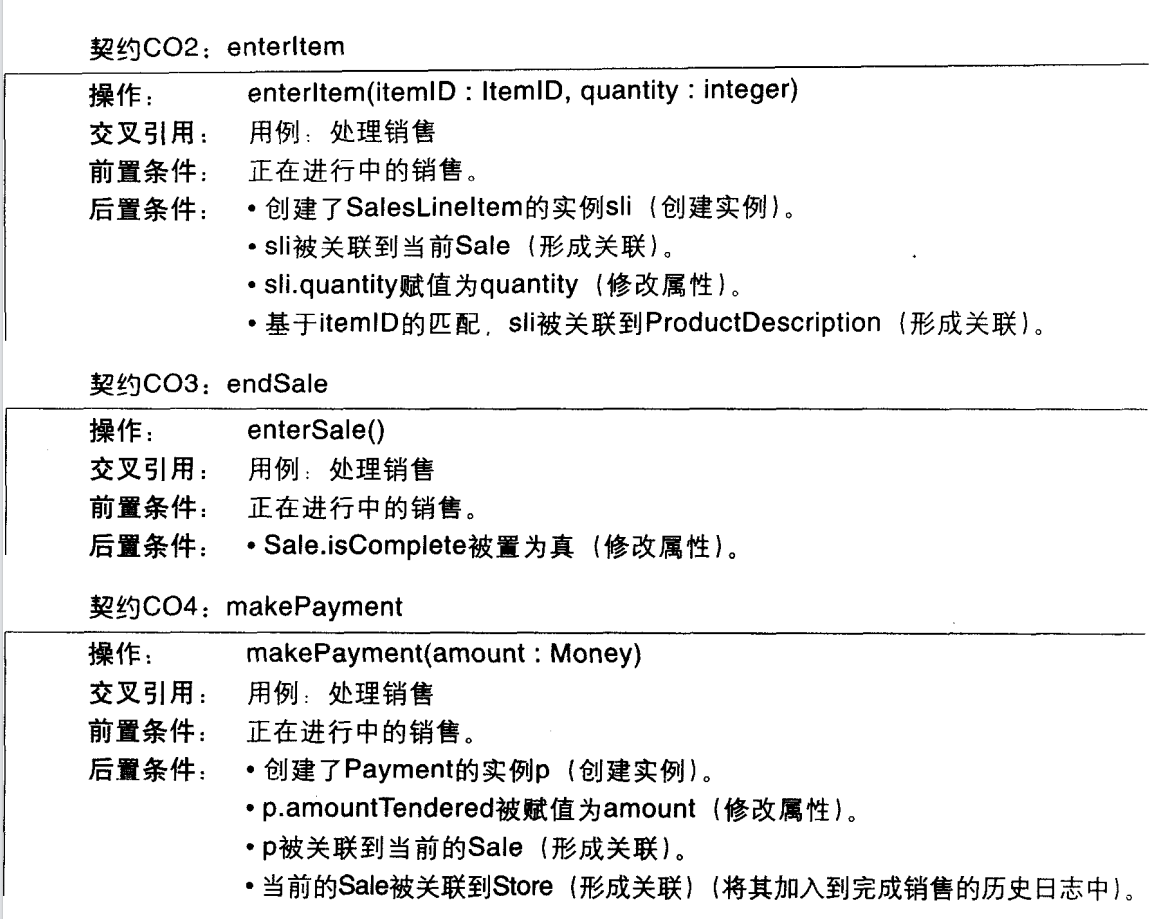
②**【注】后置条件不是在操作过程中执行的活动，而是对领域模型对象的观察结果。∴后置条件应描述为说明性的、被动的过去时态的语句。**

③后置条件分为：

创建或删除实例、属性值的变化、形成或消除关联。

e.g.注意对实例的命名——sli，这样可以简化在其他后置条件语句中对该新实例的引用。



④最后，后置条件并不总是需要的，而且也不需要为所有的系统操作生成完整详细的后置条件；后置条件支持细粒度的细节和精确性；如果在思考操作契约过程中有新的发现，那就需要对领域模型进行改进（~在迭代和进化式方法中，所有分析和设计制品都不是完善的，要根据新发现对其进行改进）。

3.操作v.s. 方法

（1）操作（operation）是可以调用对象执行的转换或查询的规格说明。

①【注】操作是抽象不是实现

②操作具有**特征标记**——名称和参数

方法是操作的实现，规定了与操作关联的算法或过程。

======================以下是设计部分===================

**第十三章 逻辑架构和UML包图**

【小结】UML包图可以作为设计模型的一部分，用来描述LA（logical architecture，逻辑架构）。UML包图的输入是补充性规格说明中记录的架构方面的约束和要点。

**一、逻辑架构（logical architecture，LA）**

1.定义：**逻辑架构**是**软件类**的宏观组织结构，将软件类组织为**包（或命名空间）、子系统和层**。之所以称之为逻辑结构，是因为其并未决定如何在不同的操作系统进程或网络中物理的计算机上对这些元素进行部署（后面说的是部署架构deployment architecture）。

~~LA定义了包，包中有关于软件类的定义。

**2.层 （Layer）**

（1）定义：层是对类、包或子系统的甚为粗粒度的分组，具有对系统主要方面加以**内聚**的职责。

（说明：每一层都代表了系统中一个特定的关注点或责任区域，而层内部的具体实现细节对于其他层来说是隐藏的。这样的分组有助于降低系统各部分之间的耦合，并提高内聚性）

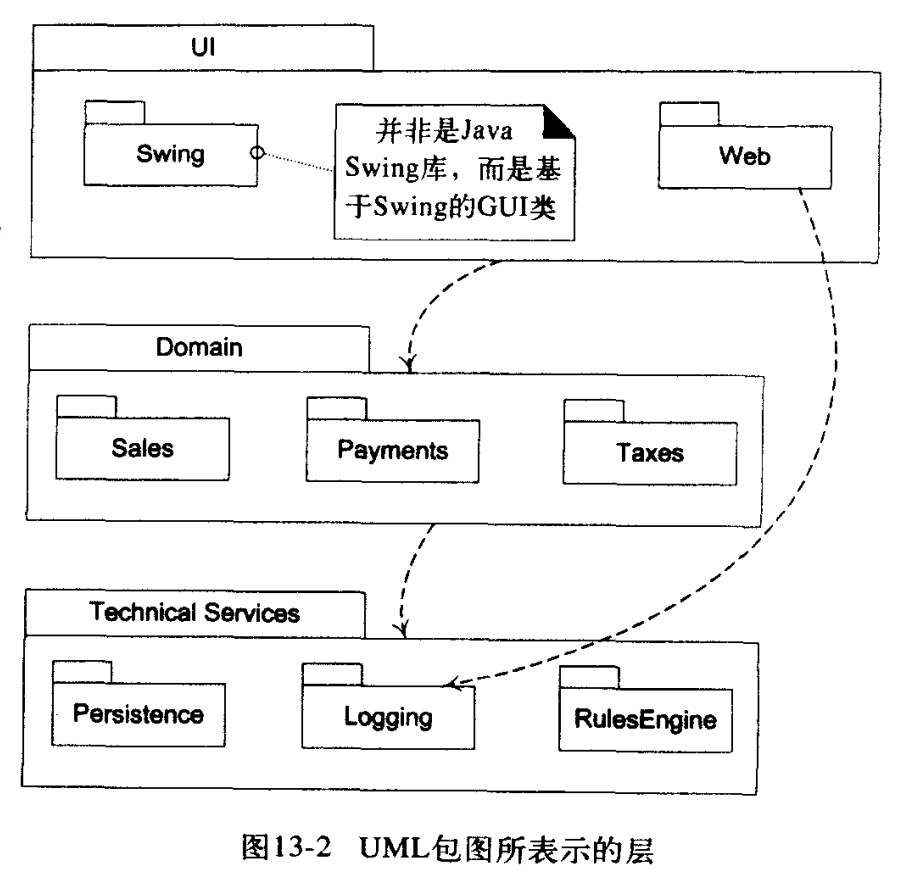
**（2）高层可以调用它下面的所有层，但是低层不能调用它上面的层**

（3）OO系统中的层通常包括：

①用户界面

②应用逻辑和领域对象——表示领域概念的**软件对象**，实现了应用需求，如下图中的Sales实现了计算销售总额。

③技术服务——提供支持性技术服务，e.g.数据库接口或错误**日志（logging**）。通常独立与应用，即可以在多个系统中复用。

3.**软件架构/架构：**

定义：架构是一组重要决策，其中涉及软件系统的组织，对结构元素及其组成系统所籍接口的选择，这些元素特定于其相互协作的行为，这些结构和行为元素到更大规模的子系统的组成，以及知道该组织结构（这些元素及其接口、协作和组成）的架构风格。

（补充：所有软件架构定义的共同主题是，必须与宏观事物有关——动机、约束、组织、模式、职责和系统之连接（系统之系统）的重要思想。

**第十五章 UML交互图（顺序图和通信图）**

**一、UML交互图（interaction diagram）**

1.定义：用于描述对象间通过消息的交互。可用于动态对象建模

2.**UML交互图有两种类型：顺序图（sequence diagram）和通信图（communication diagram）。**

【注】这里的顺序图和之前讲的系统顺序图（SSD）不同：

这里的顺序图是对类中具体方法的可视化表述；

而SSD是对用例中参与者与系统交互行为的可视化表述

**二、顺序图（sequence diagram）**

1.顺序图以一种栅栏格式描述交互，在右侧添加新创建的对象（如下图）

下图在代码中的表示：类A具有名为doOne的方法和类型为B的属性。同时类B具有名为doTwo和doThree的方法。

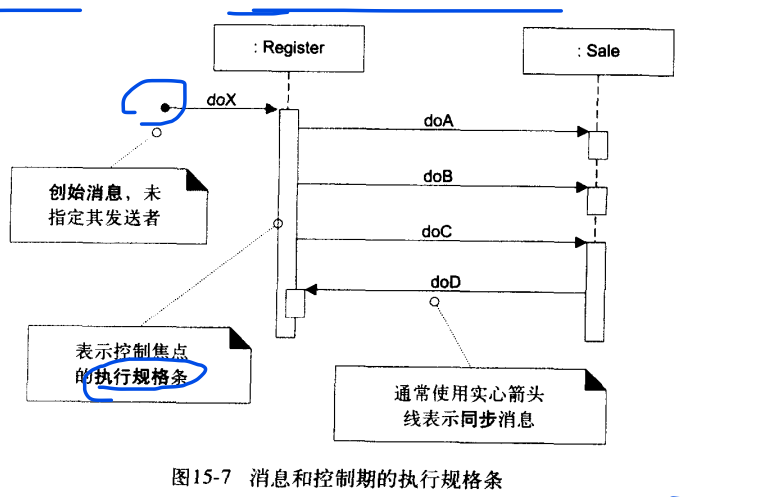
2.顺序图的基本表示法（见下图）

（1）创始消息found message：最开始的消息，在图中表示为实心圆作为起点。

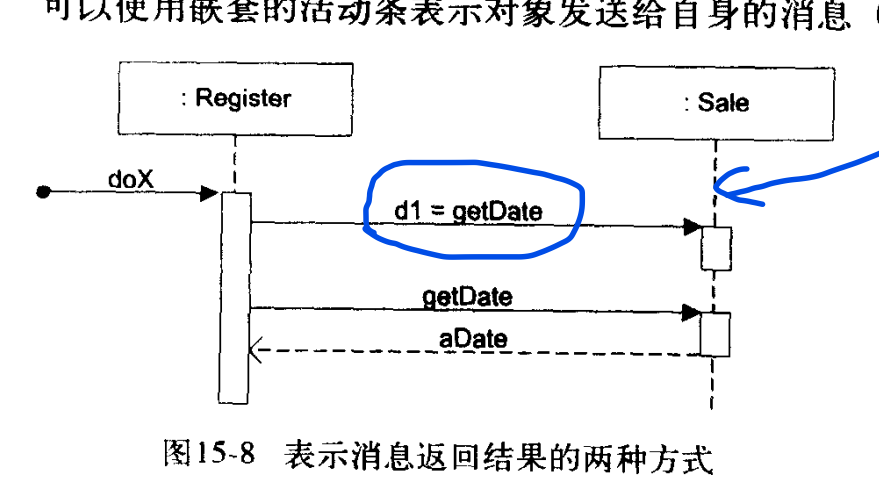
（2）①**实心箭头**+消息表达式的方式=>**同步**消息。

②**刺形箭头=>异步**调用（不是虚线，而是实线+刺形箭头）

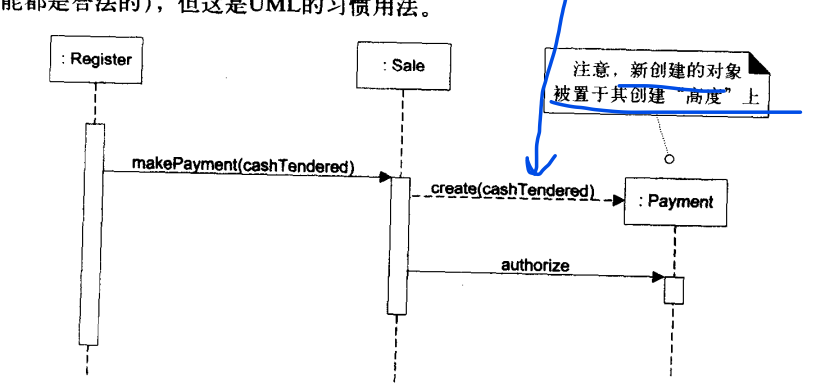
（3）生命线自上而下表示时间顺序。

（4）执行规格条execution specification bar/活动条/活动：表示**控制期focus of control**

（5）表示应答或返回的常用的两种方式：

①使用消息语法returnVar=message（parameter）

②在活动条末端使用应答（或返回）消息线

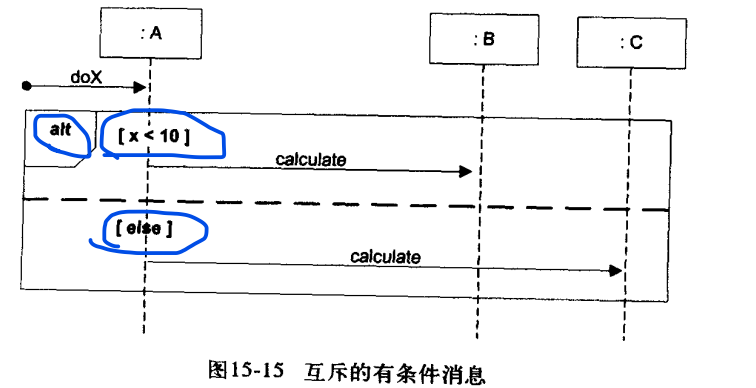
（6）**【注】创建对象时必须用虚线**

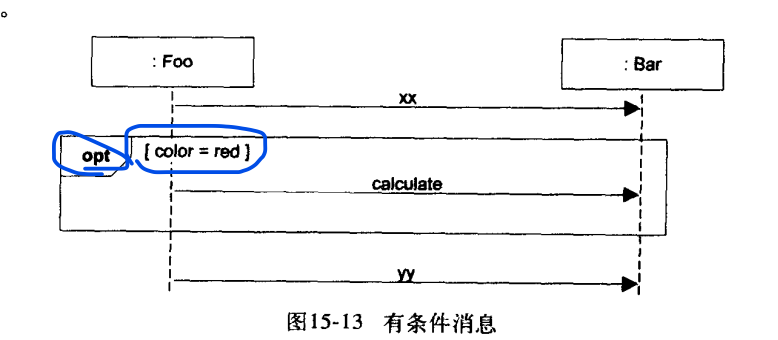
3.顺序图中的图框frame

图框是图的区域或片段，在图框中有操作符或标签（如loop）和保护信息

~~~常见的图框操作符：

①alt：选择性的片段，用于表示保护信息所表达的**互斥**条件逻辑

②loop：用于表示保护信息为真的循环片段。loop（n）表示循环的次数。也可以loop（i，1,10）

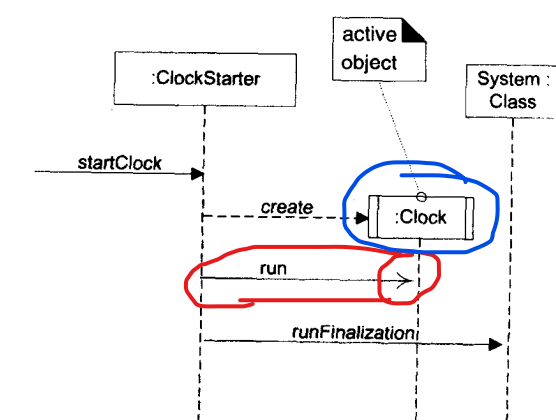
③opt：当保护信息为**真**时执行的可选片段

④par：并行执行的并行片段

⑤region：只能执行一个线程的临界片段

4.**主动对象（active object）**

（1）主动对象即在其自己的执行线程中运行或控制自己的执行线程的实例。

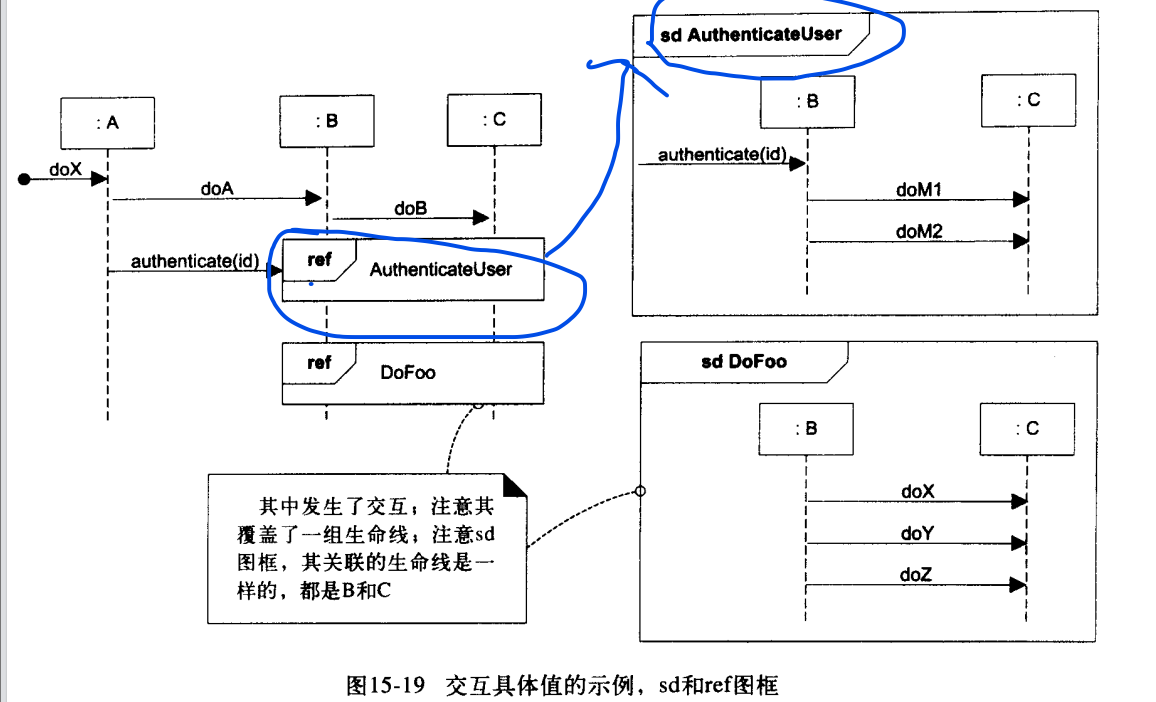
（2）表示：在生命线框图的两侧加竖线。

5.关联交互图

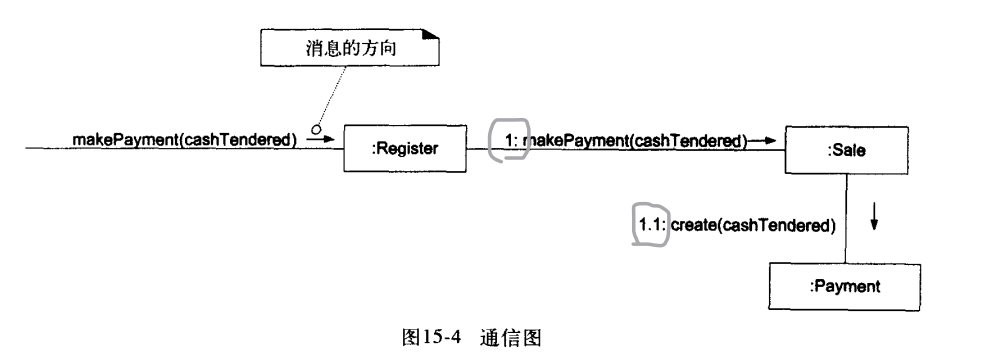
（1）交互具体值（interaction occurrence）/交互使用（interaction use）：是在交互中引用另一交互。

（2）表示：

①在整个顺序图周围放置图框，并加上sd标记和注入AuthenticateUser这样的名称。

②标记为ref的图框称为引用，该引用指另一个已命名顺序图。

**三、通信/讯图（communication diagram）**

1.通信图以图或网格格式描述对象交互，其中对象可以置于图中的任何位置（如下图）

2.在通信图中，使用顺序编号，如“1：”表示调用流的顺序。

3.**链（link）**

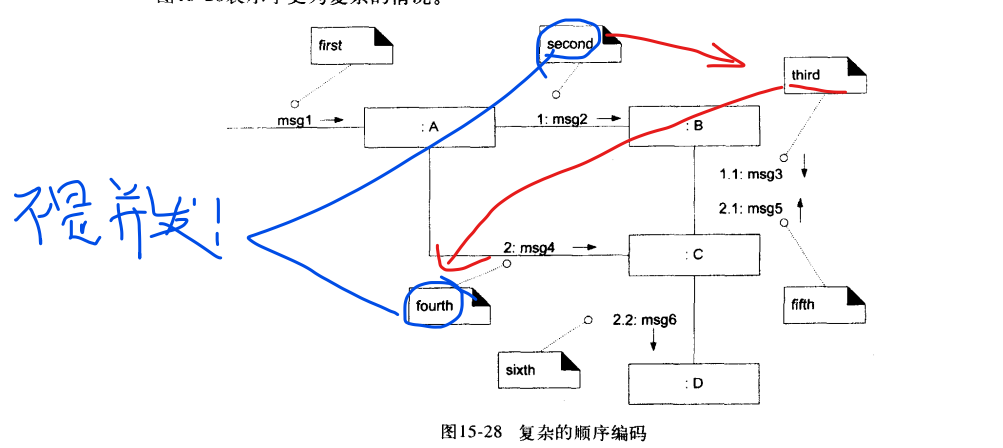
（1）链是连接两个对象的路径，指明了对象间某种可能的导航和可见性。

（2）**链是关联的实例**。

（3）**多个消息共用一条单链传输消息**~每条链都允许双向消息传输。

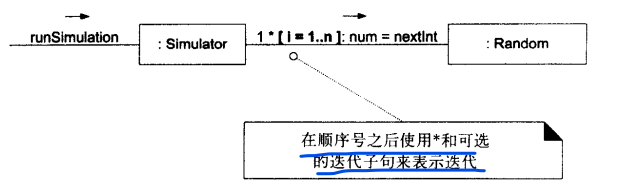
4.消息

（1）表示：**消息表达式+小箭头**（用于指明消息的方向）

（2）可以增加顺序编号以表示当前控制线程中消息的次序。

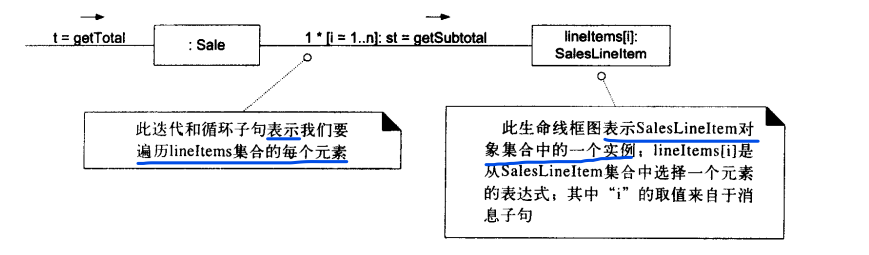
①不要为起始消息编号（虽然合法）

②有条件消息：在顺序编号后加带方括号的条件子句，e.g.“1 [color=red]:”

③互斥的有条件路径：使用条件路径字母修改顺序编号，e.g.“1a[color=red]:”

④迭代

⑤集合的迭代，如下图：

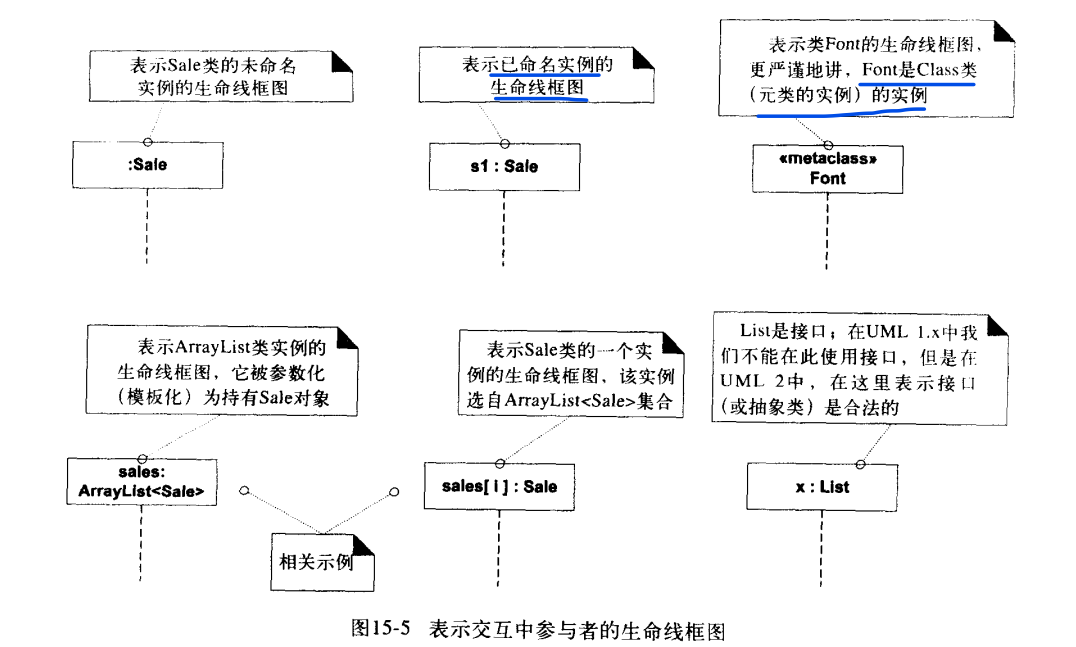
其中“1\*[i=1..n]:”表示遍历lineItem集合中的每个元素

顺序图 v.s. 通讯图：

（1）顺序图能够清晰地表示消息的顺序和时间排序，大量详细表示法选项；但是顺序图强制在右侧增加新对象->消耗水平空间

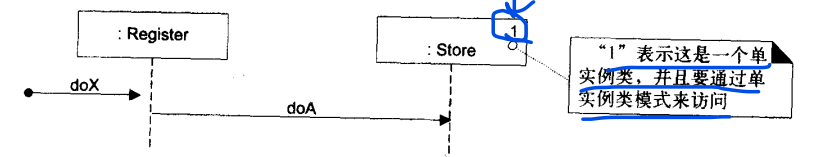
（2）通讯图的优势在于空间效用->能够在二维空间内灵活的增加对象；但是通信图不易查阅消息的顺序，而且表示法选项较少。

**四、常用的UML交互图表示法**

（1）前面交互图实例中的框图——生命线（lifeline）框图，表示的是交互的参与者（participant），注：生命线框图并不等同于类的实例。

**———————五、单实例类对象（Singleton）—————**

1.**单实例类（Singleton）模式指对类进行实例化时，只能存在一个实例，而绝不能是两个。**

2.在UML交互图中，**单实例类对象的生命线框图右上角要表示“1”**——含义：**使用单实例类模式可以获得对象的可见性**。

**第十六章 UML类图（class diagram）**

**一、类图classdiagram**

1.类图表示类、接口及其关联->用于静态对象建模，并不特定于透视图（概念或软件）

**2.类元（classifier）**

（1）定义：类元是描述行为和结构特性的模型元素，是对众多UML元素的泛化，这些元素包括类、接口、用例和参与者。

**二、设计类图（design class diagram，DCD）**

1.设计类图是使用在软件透视图和设计透视图中的类图的建模术语。

2.设计模型：包括所有DCD、UML交互图、包图。

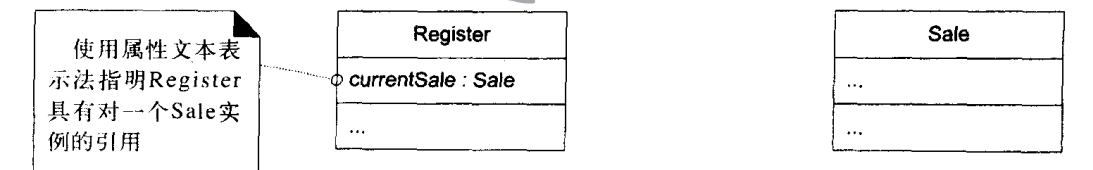
**三、表示UML属性的方法：属性文本和关联线**

1.结构化特性（structural property）：即类元的属性。

**（一）属性文本表示法的完整格式：**

可见性 名字：类型 多重性=默认值 {特性字符串}

（1）如果没有给出可见性，默认属性是私有的。

e.g.

**（二）关联线表示属性：**

（1）**导航性箭头（navigability arrow）**：由源对象指向目标对象，表示源对象的一个属性是目标对象

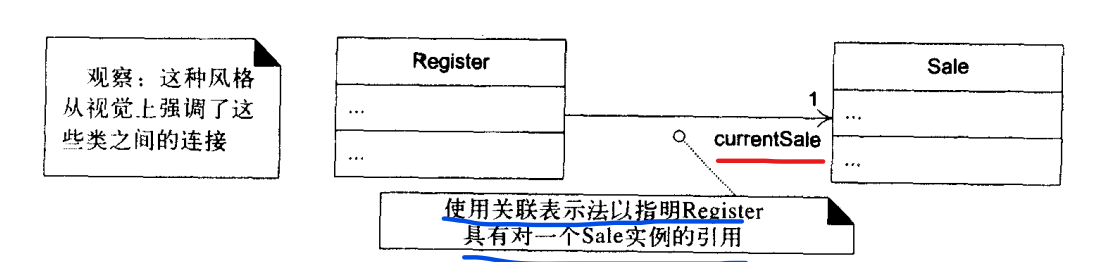
**【注】别忘了设计类图中的关联线要画刺形箭头！！！！**

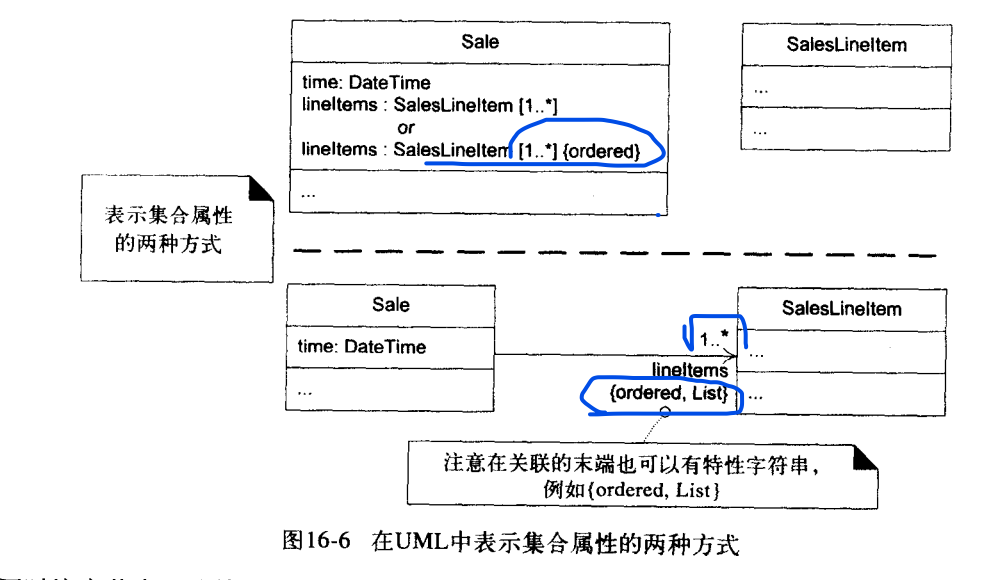
（2）【注】关联线表示属性时，多重性和角色名（rolename）只放在**目标**一端（但是放在源端也合法）

①**角色名（rolename）/关联端点名（assiciation end name）:**用于表示**属性名称**。

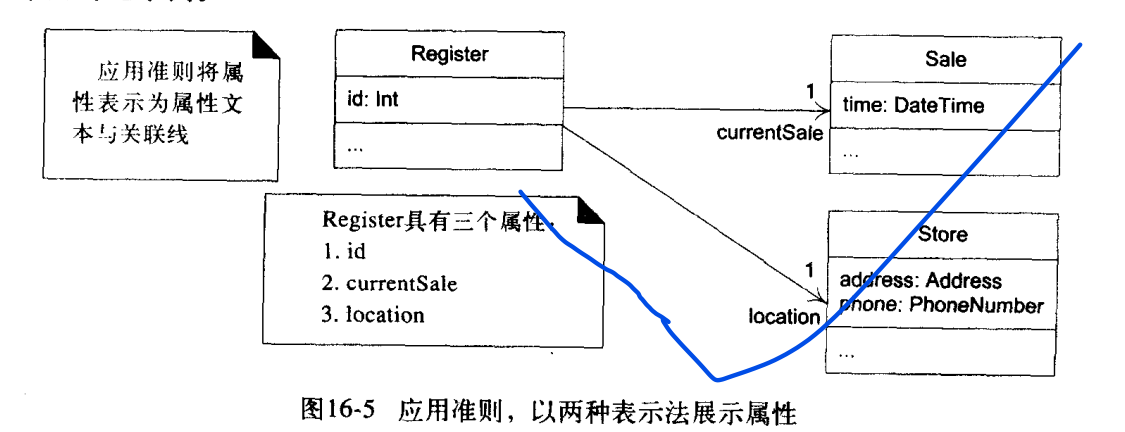
（3）【注】与领域模型不同（领域模型类图中的关联需要指明关联名称），设计类图中的关联不需要关联名称（但是有也合法）。

e.g.

（4）特性字符串：{ordered}表示集合中的元素是有序的；{unique}表示一组唯一元素。

e.g.

2.准则：对数据类型对象使用属性文本表示法，对其他对象使用关联线。

e.g.

四、其他内容

1.（1）注解符号（note symbol）：表示为折角矩形，并用虚线连接到要注解的元素上

（2）UML约束（constraint）：必须用“{}”括起来

2.操作和方法

（1）操作的语法形式：

可见性 名字 （参数列表）：返回类型 {特性字符串}

~特性property

【注】如果操作没有注明可见性，默认操作是公共的！！！（这点不同于属性）

（2）操作≠方法

操作是**声明**，其中包含名称、参数、返回类型、异常列表、可能的前置和后置条件约束等。总之，操作不是实现，而方法是实现。

（3）方法在类图中的表示：通过《method》注解符号定义方法体

~~当使用UML注解来表示方法时，实际是在同一个图中混合了动态视图（方法体定义了动态行为）和静态视图（UML类图）。

（4）交互图中的create消息通常被映射为构造器的定义；通常对构造函数冠以《constructor》

（5）**访问操作（即get和set）通常不包含在类图中**

3.**关键字（keyword**）

（1）定义：关键字是**对模型元素分类的文本修饰**。

（2）大部分关键字使用《》表示，有些关键字使用{}表示。

e.g.定义类元框图类别为接口的关键字是《interface》；

定义类元是参与者的关键字是《actor》；

表示具有强制顺序的一组对象的关键字是{ordered}；

表示是抽象元素，不能实例化的关键字是{abstract}

**4.构造型（stereotype）**

（1）定义：构造型表示**对现有建模概念的精化**，并且定义在UML简档（profile）中。

~简档（profile）是一组相关构造型、标记和约束的集合，其目的是使用UML专用于特定领域或平台。

（2）表示：与关键字一样，也是使用《》表示。

（3）UML预定义了大量构造型，同时允许用户自定义构造型（∴UML中构造型提供了扩展机制）

（4）构造型使用属性语法声明一组标记（tag）。

5.泛化generalization：

（1）定义：泛化是普通类元与特殊类元之间的分类学关系。特殊类元的每个实例也是普通类元的间接实例。特殊类元间接地拥有了普通类元的特性。

（2）表示：子类到超类的实线+空心三角箭头

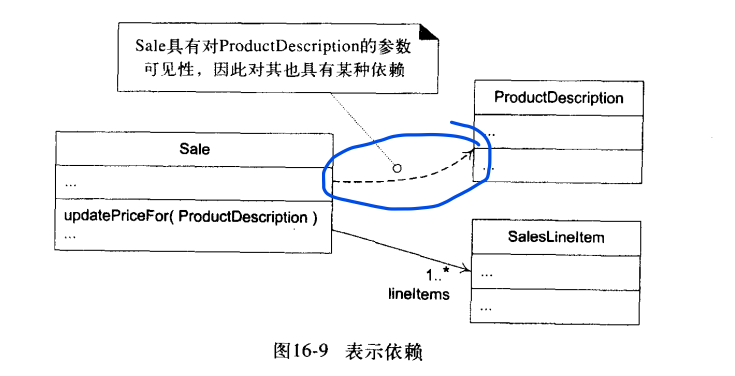
**五、依赖（dependency）**

1.依赖关系表示**客户元素**（任何种类，包括类、包、用例等）**了解其他提供者元素**，并且表示**当提供者有所改变时会对客户产生影响**。

~可以将依赖看成是耦合的另一个版本，即某元素耦合或依赖于另一元素。

2.表示：从客户到提供者的虚线箭头线

~依赖线可以用于任何图形。

e.g.

3.常见的依赖类型：

（1）拥有提供者类型的属性。

（2）向提供者发送消息。对提供者的可见性可能是：属性、参数变量、局部变量 、全局变量或类的可见性（调用静态或类方法）

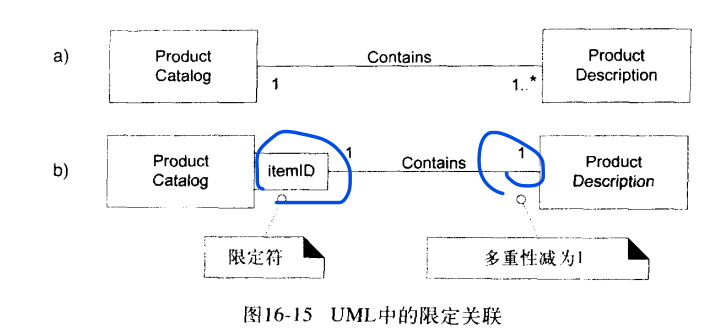
（3）接收提供者类型的参数。

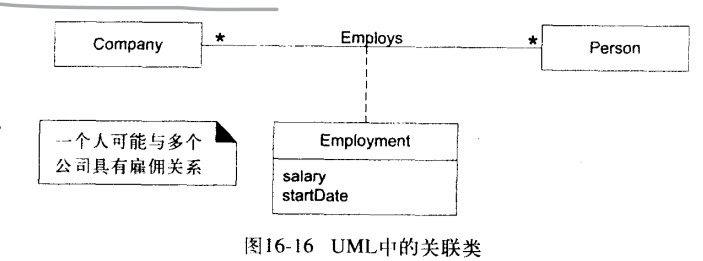
（4）提供者是超类或接口。

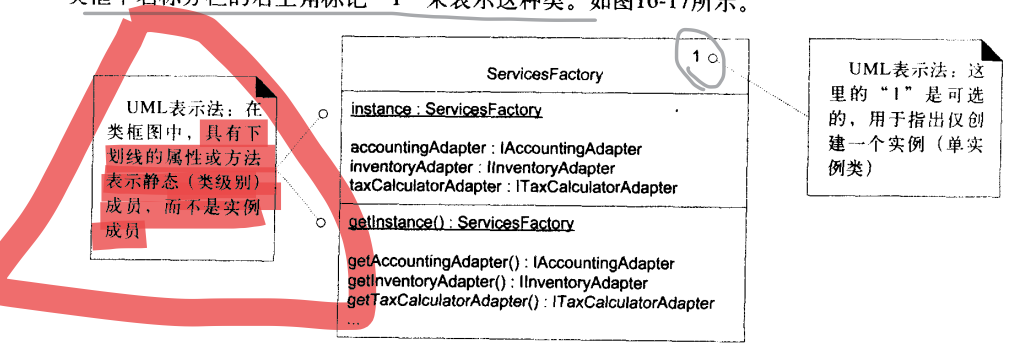
4.可以给依赖线附加关键字或构造型以表示依赖类型。~~称为依赖标签

5.约束constraint：用花括号表示

6.限定关联（qualified association）具有限定符（qualifier），限定符用于从规模较大的相关对象集合中，依据限定符的键选择一个或多个对象。

~~对于限定关联，要注意多重性的变化，通常是多变一

7.关联类（association class）：允许将关联本身作为类，并且使用属性。操作和其他特性对其建模。

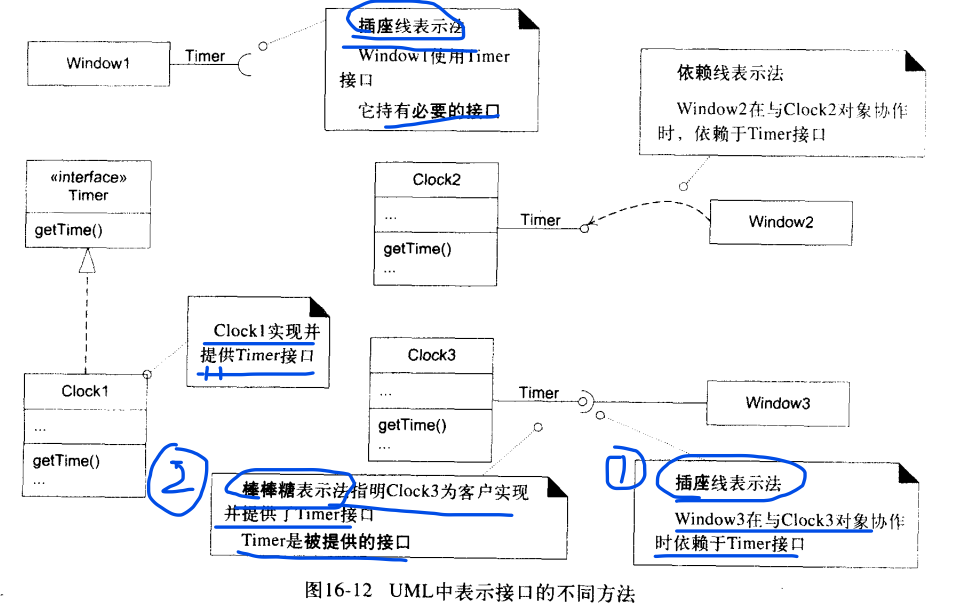
8. **在UML类框图中，具有下划线的属性或方法表示静态（类级别）成员，而不是实例成员！！！！！！！！！！**

**六、接口（interface）**

1.接口包括客户的接口和接口依赖（必要接口）。

2.接口实现（interface realization）

（1）插座线表示法（socket notation）：表示“类X需要（使用）接口Y”

（2）棒棒糖表示法：表示提供接口

**七、组合（composition）优于聚合（aggregation）**

（一）聚合（aggregation）——一种模糊的关联

**（二）组合（composition）/组成聚合（composite aggregation）**

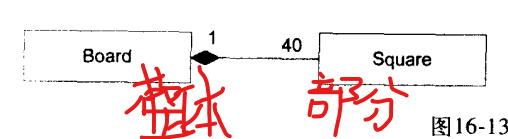
1.组合关系的含义：

（1）在某一时刻，部分的实例只属于一个组成实例；

（2）部分必须总是属于组成，即**不允许存在游离的部分！！！**

（3）组成要负责创建和删除其部分，（可以自己创建/删除，也可以和其他对象协作创建/删除部分）∴如果组成被销毁，其部分也必须被销毁，或者依附于其他组成。

2.**组合的表示：实心菱形箭头，菱形位于整体一侧。**



**第十七章 GRASP：基于职责设计对象**

GRASP（general responsibility assignment software patterns）通用职责分配软件模式

【注】在图绘制UML期间，我们绘制模型主要是为了理解和沟通，并不是为了编写文档。

**一、职责驱动设计（RRD）**

1.职责驱动设计（RRD）：强调根据对象的职责（即它们应该做什么）来定义对象

。思考软件对象设计以及大型构件的流行方式是烤炉其职责、角色和协作。

2.职责

（1）定义：职责是类元的契约或义务。

（2）类型：职责分为行为职责和认知职责。

①行为职责包括：自身执行一些行为，e.g.创建对象/计算；初始化其他队吸纳高中的动作；控制和协调其他对象中的活动

②认知职责包括：对私有封装数据的认知；对相关对象的认知；对其能够导出或计算的失误的认知。

3.RDD包括了协作的思想，即职责进驻与方法来实现，该方法既可以单独动作，也可以与其他方法和对象协作

4.职责与方法并非同一事物，职责是一种抽象，而方法实现了职责。

5.当我们绘制UML交互图时，就是在决定职责的分配。

**———————————二、GRASP——————————**

1.**模式**的定义：有经验的OO开发者（以及其他的软件开发者）建立了既有通用原则又有惯用方案的指令系统来指导他们编制软件。如果以结构化形式对这些**问题、解决方案和命名**进行描述使其系统化，那么这些原则和习惯用法就可以成为模式。

~~模式具有名称。GRASP模式陈述的并不是新思想，而只是为广泛使用的进本原则命名并将其汇总起来。

**（一）创建者（creator）模式**

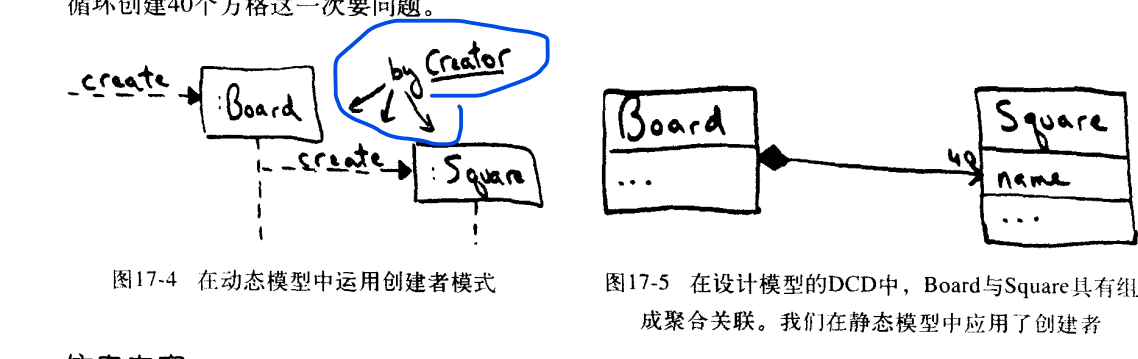
1.定义：

名称：创建者（creator）

问题：**谁创建了A？**

解决方案（可被视作建议）：如果一下条件之一为真时（越多越好），将创建类A实例的职责分配给B：

B“包含”或组成聚集了A、B记录A、B紧密地使用A、B具有A的初始化数据。

【注】A和B指定是软件对象，而不是领域模型对象。

**（二）信息专家（information expert）**

1.

**（三）低耦合（low coupling）模式**

**1.耦合是对某元素与其他元素之间的连接、感知和依赖程度的度量。具有低耦合的元素不会过度依赖于其他元素。**

**（四）高内聚（high cohesion）模式**

**1.内聚（更专业的说，是功能内聚）是对元素职责的相关性和集中度的度量。如果元素具有高度相关的职责，且没有过多工作，那么该元素具有高内聚性。**

**（五）控制器（controller）模式**

**1.定义：控制器是UI层之上的第一个对象 ——负责接受和处理系统操作消息。控制器模式考虑的问题是：在UI层之上首先接收和协调（控制）系统操作的第一个对象是什么。**

2.解决方案：把职责分配给能代表一下选择之一的类：

（1）代表整个“系统”、“根对象”、运行软件的设备或主要子系统，这些事外观控制器的所有变体。

（2）代表用例场景，在该场景中发生系统事件，通常命名为<UseCaseName>Handler或<UseCaseName>Coordinator或<UseCaseName>Session。

（3）对于同一用例场景的所有系统事件使用相同的控制器类。

（4）会话是与参与者进行交谈的实例。会话可以具有任意长度，但通常按照用例来组织（用例会话）

**第二十五章 GRASP**

（六）多态（polymorphism）模式

（七）间接性（indirection）模式

（八）纯虚构（pure fabrication）模式

（九）防止变异（protected variation）模式

**1.开放-封闭原则（OCP，open-closed）**

**即模块应该同时（对扩展、自适应）开放和（对影响客户的更改）封闭。即OCP提倡在软件开发过程中通过添加新代码（扩展）来实现新功能，而不是通过修改已有代码来实现新需求。基本上等价于PV模式和信息隐藏。**

**第二十六章 GoF模式**

（一）适配器模式（adaptor）

（二）工厂模式（factory）

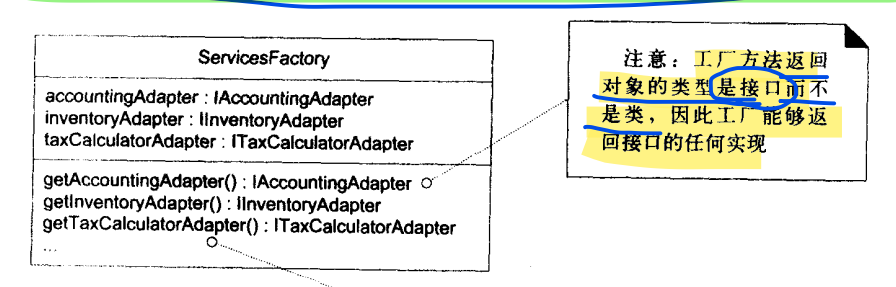
1.适配器引起的问题：谁创建什么适配器？

解决：定义纯虚构的“工厂”对象来创建对象。

2.名称：工厂（factory）

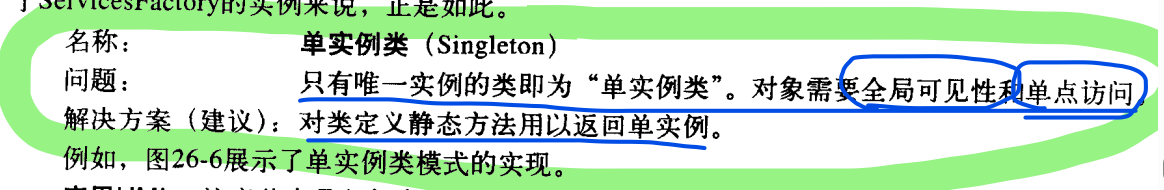
问题：当有特殊考虑（如存在复杂创建逻辑、为了改良内聚而分离创建职责等）时，应该由谁来负责创建对象？

解决方案（建议）：创建称为工厂的纯虚构对象来处理这些创建职责。

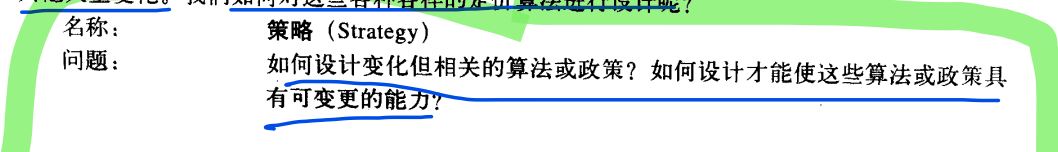
【注】工厂方法返回对象类型是接口不是类，∴工厂能够返回接口的任何实现。

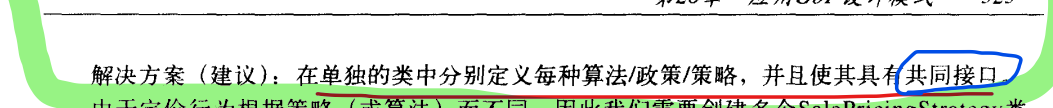
（三）单实例类(Singleton)

工厂模式带来的问题：谁创建工厂，如何访问工厂？

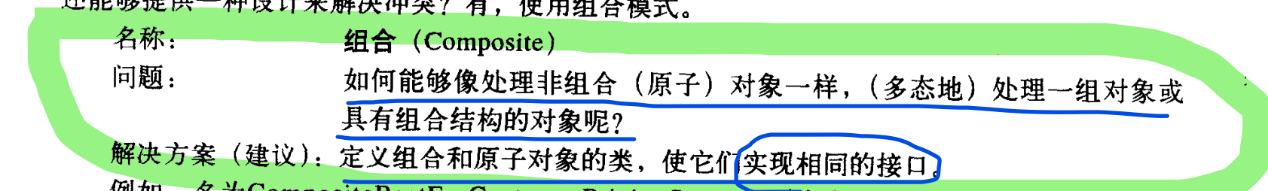
解决：单实例类模式（∵在该过程中只需要一个工厂实例）

（四）策略模式

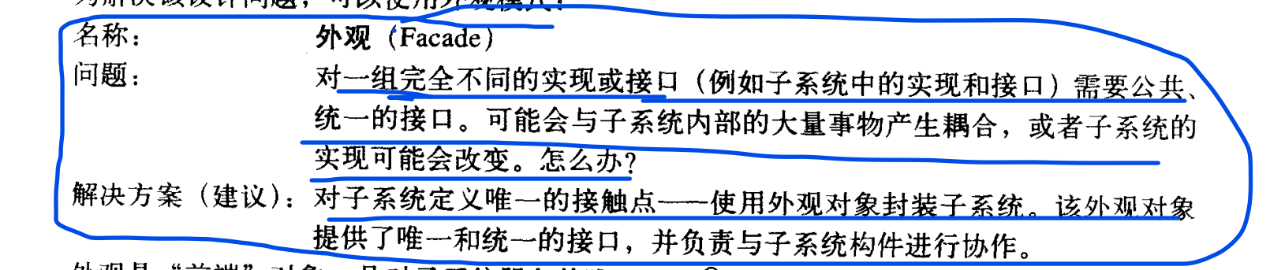
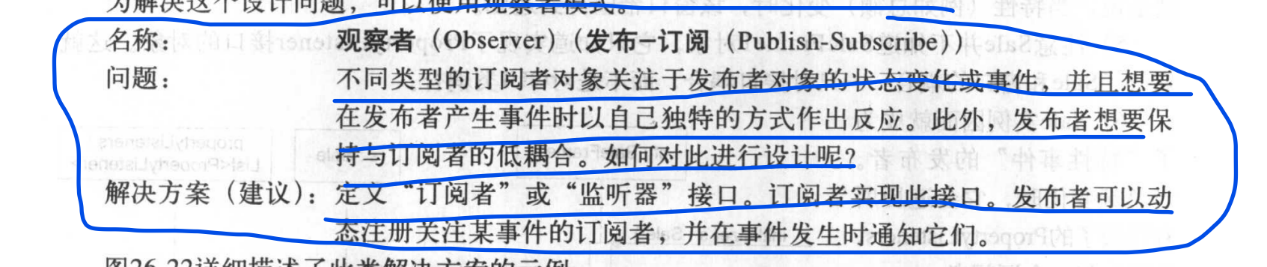
∵销售的定价策略具有多样性。

****

（五）组合（composite）

****解决冲突的策略

（六）外观模式（facade）

（七）观察者observer/发布-订阅（publish-subscribe）

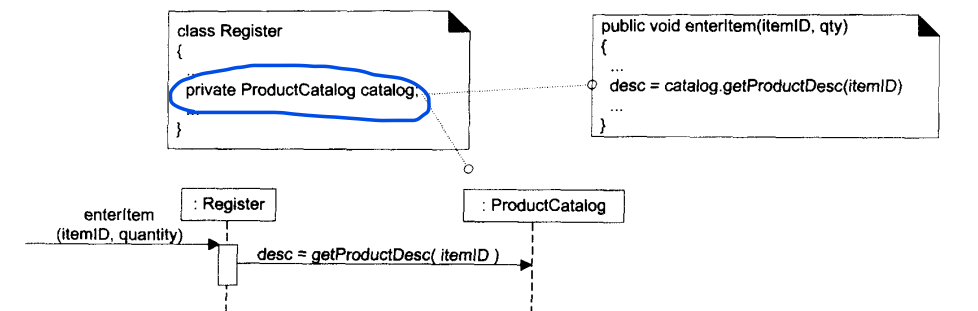
**第十九章 可见性**

1.可见性（visibility）：

定义：**可见性是一个对象看见或引用其他对象的能力**。

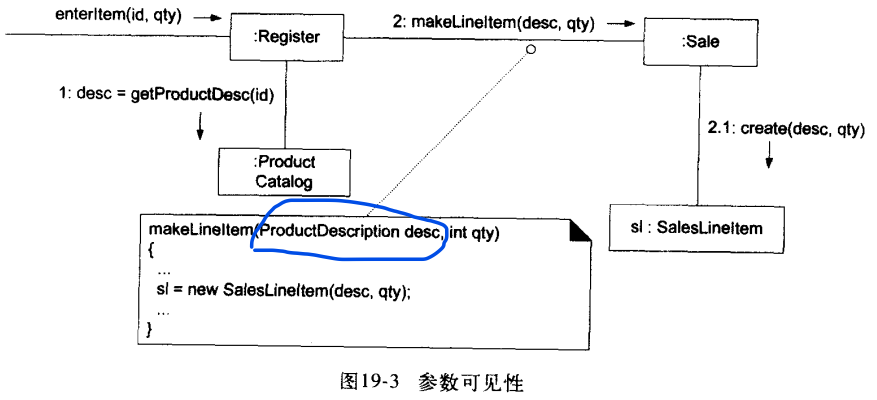
2.准则：**为了使对象A可以向对象B发送消息 ，对于A而言，B必须是可见的。**

（一）属性可见性（attribute visibility）

当B作为A的属性时，存在A到B的属性可见性——相对持久

e.g.

（二）参数可见性（parameter visibility）

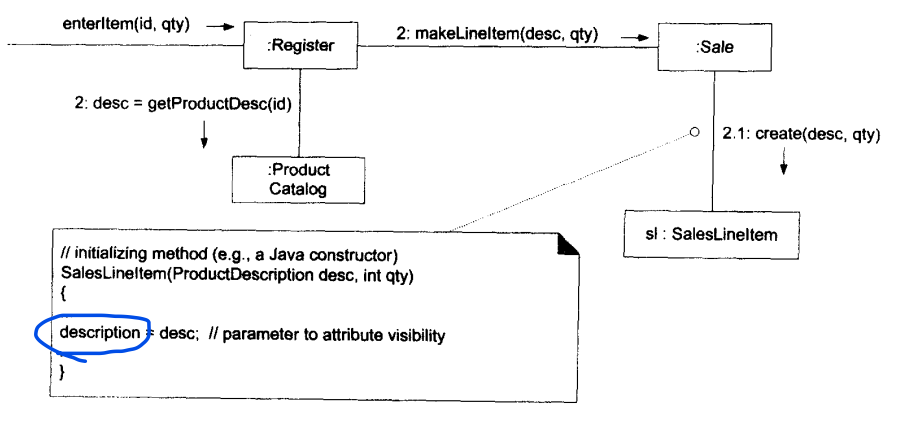
（1）当B作为参数传递给A的方法时，存在A到B的参数可见性——相对短暂（∵只在方法的范围内存在）

e.g.

（2）可以将参数可见性转化为属性可见性。E.g.在上图中，Sale将ProductDescription传递给SalesLineItem的初始化方法，在初始化方法中，将参数分配给属性，以此建立起属性可见性。

（三）局部可见性（local visibility）

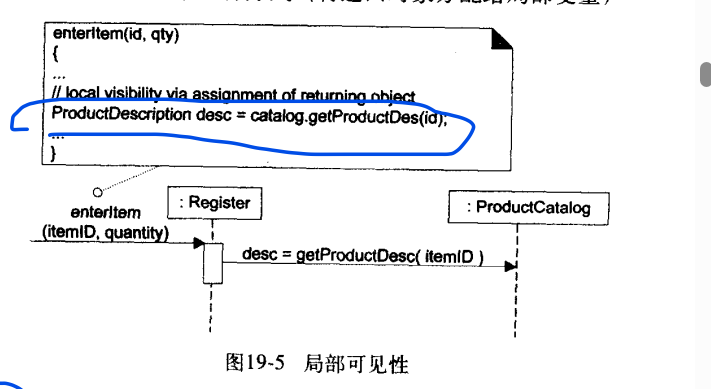
1.当B被声明为A的方法内的局部对象时，存在A到B的局部可见性——相对临时（∵仅存在于方法的范围内）

e.g.

2.实现局部可见性的两种常见方式：

（1）创建新的局部实例并将其分配给局部变量；

（2）将方法调用返回的对象分配给局部变量。

e.g.

（四）全局可见性（global visibility）

1.当B对于A是全局时，存在A到B的全局可见性——相对持久

2.实现全局可见性的首选方法是使用单实例类模式

**第三十章 用例关联**

**引言：**

（1）具体用例（concrete use case）：由参与者发起，完成了参与者所期望的完整性为。E.g.处理销售

（2）抽象用例（abstract use case）：永远不能被自己实例化；不能独立存在，只能是其他用例的一部分（即其他用例的子功能用例）。E.g.处理信用卡支付//

（3）基础用例（base use case）：包含其他用例的用例，或者是被其他用例扩展或泛化的用例。E.g. 处理销售

（4）附加用例（addition use case）：被其他用例包含的用例，或者扩展、泛化其他用例的用例。E.g.处理信用卡支付//

~~附加用例通常是抽象用例；基础用例通常是具体用例。

**一、包含关系（include）**

**【小结】什么时候使用包含关系？**

**①当两个或多个独立用例中存在重复，而你想避免这种冗余时，可以使用包含关系**

**②将过于冗长的用例简单地分解为子单元可以方便理解**

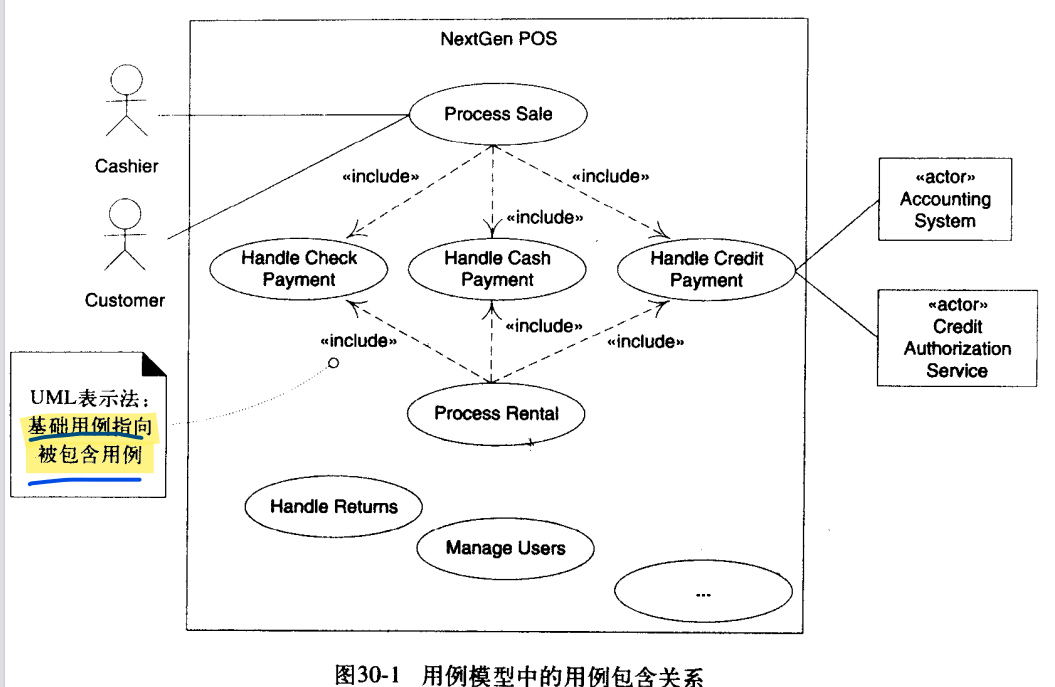
**③描述异步事件的处理时可以使用包含关系。**

1.**在用例文本中，使用下划线或高亮表示被包含的用例**

e.g.处理销售、处理租金、向失业救济计划捐款等用例可能都包含了信用卡支付的交互行为，∴可以将信用卡支付部分单拎出来作为一个子用例功能，然后用包含关系加以指示。

注：子用例也有主成功场景和扩展等结构

2.下面谈在异步事件处理中使用包含关系：

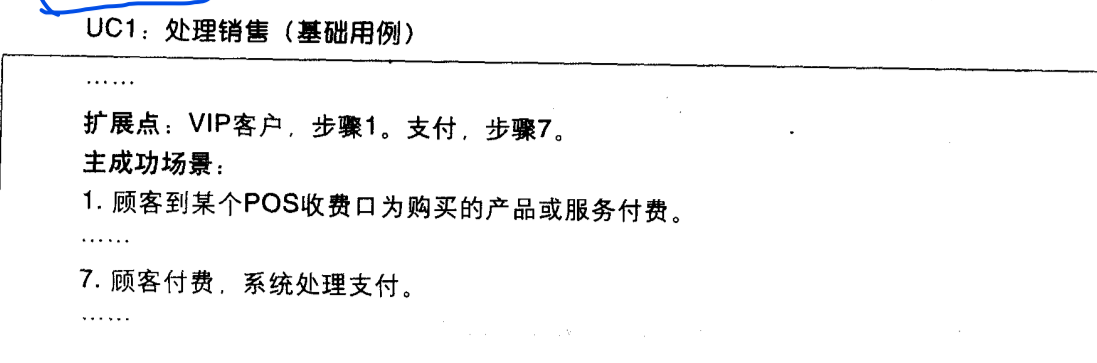
其实，也可以在扩展部分中使用像“\*a”这样的标记。

在用例图中的表示：基础用例指向被包含用例，并在虚线上标明《include》

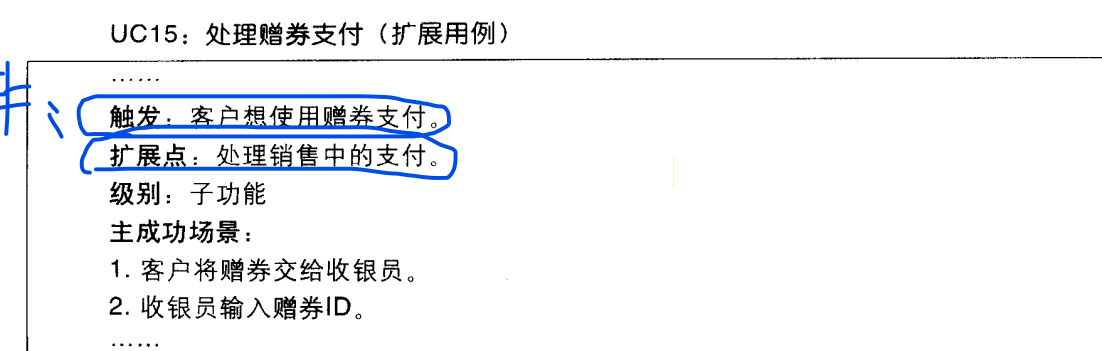
**二、扩展关系（extend）**

**【小结】当基础用例不能修改时，可以使用扩展关系**

1.思路：创建扩展或附加用例，并且在其中描述在**何处和何种条件**下该用例扩展**某基础用例**的行为

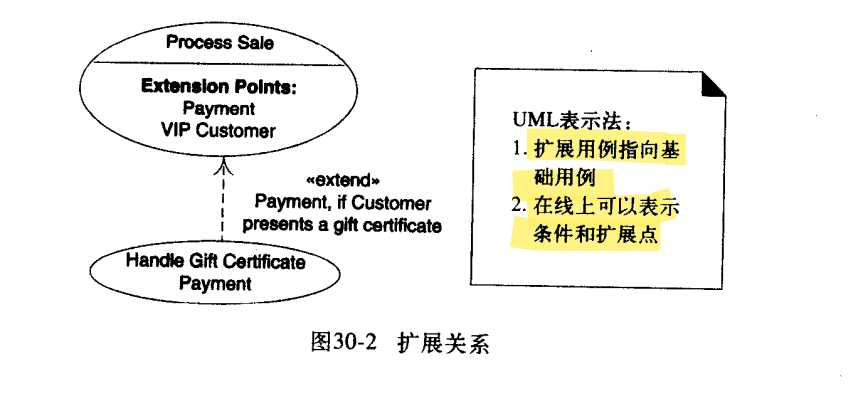
e.g.

在下图中，扩展点是基础用例的标记，扩展用例是通过该标记引用扩展点的，∴基础用例的步骤编号可以改变，而不会影响扩展用例。

2.在扩展关系中，基础用例对扩展用例没有任何引用。即基础用例不需要定义或处理扩展用例触发的条件。e.g.处理销售用例（基础用例）自身是完备的，不需要知道处理信用卡支付（扩展用例）的信息。

3.其实，处理赠券支付附加用例也可以通过包含关系在处理销售用例中引用；

或者直接将赠券支付的场景添加在处理销售用例的扩展部分。（推荐直接更新基础用例的扩展部分，而不是使用扩展关系再写一个用例）

4.扩展在用例图中的表示：扩展用例指向基础用例；在虚线上可以表示条件和扩展点

**三、泛化关系（generalization）**

类似于java中的继承，e.g.支付用例，对他泛化的用例可以是信用卡支付、现金支付，就是在支付的基础上，具体化了支付方式（继承）