**第四部分 细化迭代2 更多模式**

1、在熟练使用UP的项目中，为早期迭代所选择的需求,是根据风险和高业务价值组织的,这样就能够尽早识别并解决高风险问题。

2、最后四个GRASP模式：

**多态（Polymorphism）**

当相关选择或行为随类型（类）有所不同时，使用多态操作为变化的行为类型分配职责。不要测试对象的类型，也不要使用条件逻辑来执行基于类型的不同选择。

除非在超类中具有默认的行为，否则将超类中的多态方法声明为{abstract}。

多态意味着在大部分OO语言中要使用抽象超类或接口。何时应该考虑使用接口呢？普遍的答案是，当你想要支持多态但是又不想约束于特定的类层次结构时，可以使用接口。如果使用了抽象超类AC而不是接口，那么任何新的多态方案都必须是AC的子类，这对于诸如Java和C#的单根继承语言来说将十分局限。经验的做法是：如果有一个具有抽象超类C1的类层次结构，可以考虑对应于C1中的公共方法特征标记来定义接口I1，然后声明C1来实现接口I1。

**间接性（Indirection）**

将职责分配给中介对象，使其作为其他构件或服务之间的媒介，以避免它们之间的直接耦合。中介实现了其他构件之间的间接性（indirection）。

“计算机科学中的大多数问题都可以通过增加一层间接性来解决”，这一格言特别适用于面向对象设计。

**纯虚构（Pure Fabrication）**

对人为制造的类分配一组高内聚的职责，该类并不代表问题领域的概念-虚构的事物，用以支持高内聚、低耦合和复用。注：通常作为辅助类/帮助类使用。

**防止变异（Protected Variation）**

识别预计变化或不稳定之处，分配职责用以在这些变化之外创建稳定接口。

*幸运将会出现在规划之后。 -布兰奇.瑞基(Branch Rickey)*

**第五部分 细化迭代3 中级主题**

1、一个UML活动图表示一个过程中的多个顺序活动和并行活动。这些活动图有助于对业务过程、工作流、数据流和复杂算法进行建模。

基本的UML活动图表示法包括：

* 动作（action）- 它完成某些事物。在其完成时存在一个自动转换。
* 分区（partition）- 表示参加过程的不同参与者。
* 分叉点（fork）- 一个输入转换，以及多个输出的并行转换或对象流。
* 连接点（join）- 多个输入转换或对象流，一个输出变换，其输出直到所有输入都到达时才发生。
* 对象节点（object node）- 由动作产生或使用的对象，这允许我们对数据流或对象流进行建模。
* 决策（decision）- 互斥发生的任何分支。
* 合并（merge）- 任何输入的延续。

2、在活动图建模方面，有下面一些准则：

* 活动图通常对于涉及众多参与者的非常复杂的业务过程建模具有价值。对于简单的业务过程，用例文本就够用了。
* 在进行业务过程建模时，可以利用耙子（rake）符号和子活动图。
* 与上一条相关的是，尽量保持同一张图中所有动作节点的抽象水平一致。

3、同活动图一样，UML状态图是动态视图。UML包含了可用来描述事物（事务、用例和人等）的事件和状态的表示法。UML状态机图（state machine diagram）描述了某个对象的状态和感兴趣的事件以及对象响应该事件的行为。转换（transition）用标记有事件的箭头表示。状态（state）用圆角的矩形表示。通常的做法是会包含一个初始伪状态，当实例创建时，自动从初始伪状态转换到另外一个状态。

4、状态图显示了对象的生命周期：对象经历的事件、对象的转换和对象在这些事件之间的状态。状态图不必描述所有可能的事件；如果所发生的事件未在图中表示，则说明其不影响该状态机图所关注的内容。

5、用例彼此之间可能具有联系：

* 包含关系 - 多个用例中存在部分相同的行为，与其重复文本描述，不如将这部分交互行为分离为单独的子功能用例，并适用包含关系加以指示。Fowler给出了何时使用包含关系的简单且实用的准则：当在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。
* 扩展关系 - 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。

6、架构分析的本质是要识别影响架构的因素，理解这些因素的可变性和优先级，并且解决这些问题。其难点是要知道应该问什么样的问题，权衡利弊和了解处理一个重要架构因素的各种办法。

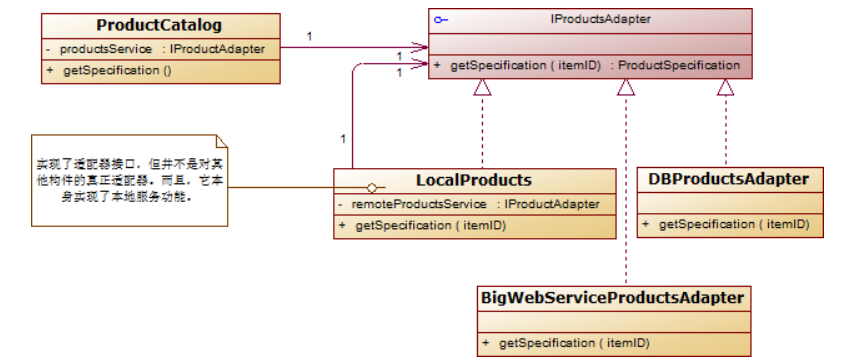
7、架构分析（architectural analysis）是在功能性需求的语境中，识别和处理系统非功能性需求的活动。

# 第28章 28.5示例：NextGen POS

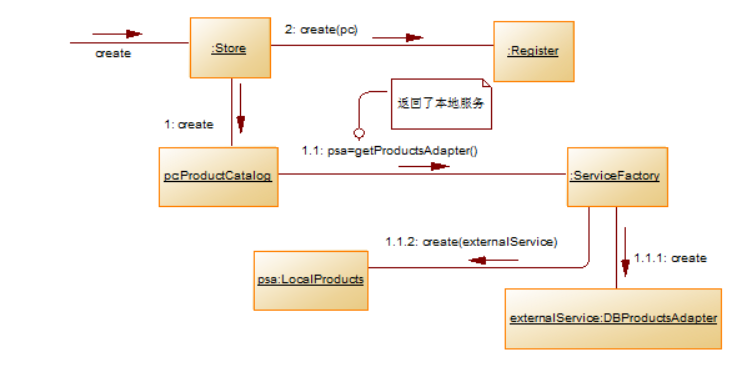
1、需求之一是，当远程服务访问失败时（例如当产品数据库（暂时）无法访问时），得到某种程度的恢复。

使用适配器和工厂模式能够间接地实现这一特性：

* ServiceFactory总是返回本地产品信息服务的适配器。
* 本地产品“适配器”并不会真正地适配其他构件，它将自己负责实现本地服务。
* 使用实际的远程产品服务适配器的引用来初始化本地服务。
* 如果本地服务在缓存中找到数据，就将数据返回；否则，将请求转发给外部服务。



对于产品信息服务的初始化：

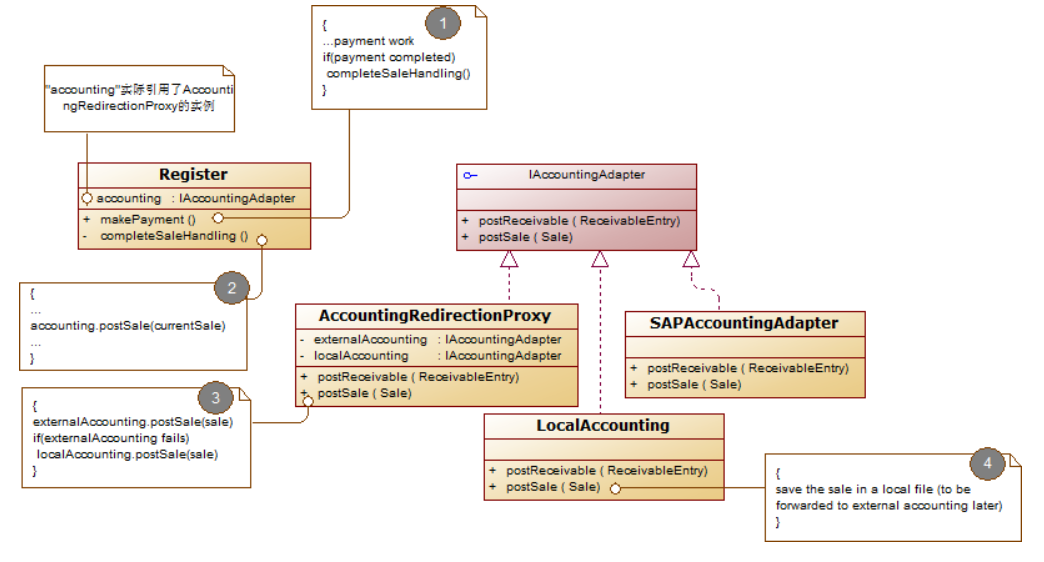


2、通过代理（GoF）使用本地服务进行容错

通过在外部服务的前端添加本地服务，实现了产品信息的本地服务容错；使用中总是优先尝试本地服务。但是，此设计方案并不是对所有的服务都适用。有时需要先尝试外部服务，然后才是本地服务。例如，在帐务服务中记录销售。在业务上希望这一过程越快越好，以便能够实时地追踪商店和终端的活动。

在此情形下，GoF的代理（Proxy）模式可以解决这个问题。NextGen案例中使用的代理是重定向代理（Redirection Proxy）变体，该代理也称为冗错代理（Failover Proxy）。

代理只不过是与被代理对象实现相同接口的对象，它保存指向被代理对象的引用，并且用于控制对被代理对象的访问。



**第六部分 其他主题**

1、有很多种方法可以计划一次迭代，下面是较典型的方法：

* 第一步应确定迭代的时间长度；常见的时间长度为2~6周。一般来说，短一些较好。延长迭代周期的因素包括在早期工作中有较多发现和变化、较大的开发团队以及分布式开发。迭代的结束时间一旦确定，就不应改变-这是时间定量的实践。但是，可以通过减少本次迭代的工作范围来满足该结束时间。
* 第二步是召集迭代计划会议。这通常在上一次迭代完成（例如星期五）而下一次迭代（例如星期一）尚未开始时进行。在理想情况下，主要涉众都应该参加：顾客（营销人员、最终用户）、开发者、首席架构师和项目经理等。
* 列出本次迭代的潜在目标（新特性或者用例、缺陷等），并标记优先级。目标列表通常由客户（业务目标）和首席架构师（技术目标）共同确定。
* 团队的每个成员应为本次迭代编制个人自愿预算（以消失或天为单位），所有的资源预算应当被汇总起来。
* 对于某一目标（例如一个用例），在计划中对其进行较为详细的描述，并分解其对应的问题。接着，参加会议的人（特别是开发人员）对与目标相关的一组更详细的任务进行头脑风暴式的讨论，并形成粗略估计。
* 反复进行第五步直到确定了足够多的工作：迭代阶段的总任务应该与总的资源预算相匹配。在指定的资源预算和时间定量最终期限的限制之下，如果工作量基本匹配，则可以结束会议。

2、迭代开发的重要思想之一就是根据反馈不断改进，而不是试图详细预测和计划整个项目。