第26章 应用GoF设计模式
Applying GOF Design Patterns

目标

- 介绍和应用一GoF设计模式
- 说明GRASP原则是对其他设计模式的归纳

GoF设计模式

GoF模式有23种，其中常用的有15种。要深入学习设计模式，请参考《Head First设计模式》

- 适配器模式（Adapter）
- 工厂模式（Factory）
- 单实例类模式（Singleton）
- 策略模式（Strategy）
- 组合模式（Composite）
- 外观模式（Facade）
- 观察者模式（Observer）

适配器模式（Adapter）

模式名：适配器（Adapter）

问题：如何解决不相容的接口问题，或者说如何为具有不同接口的相似组件提供一个稳定的接口？

解决方案：通过一个中间的适配器对象，使一个组件的原有接口转变成另一个接口。

NextGen POS中使用适配器的例子

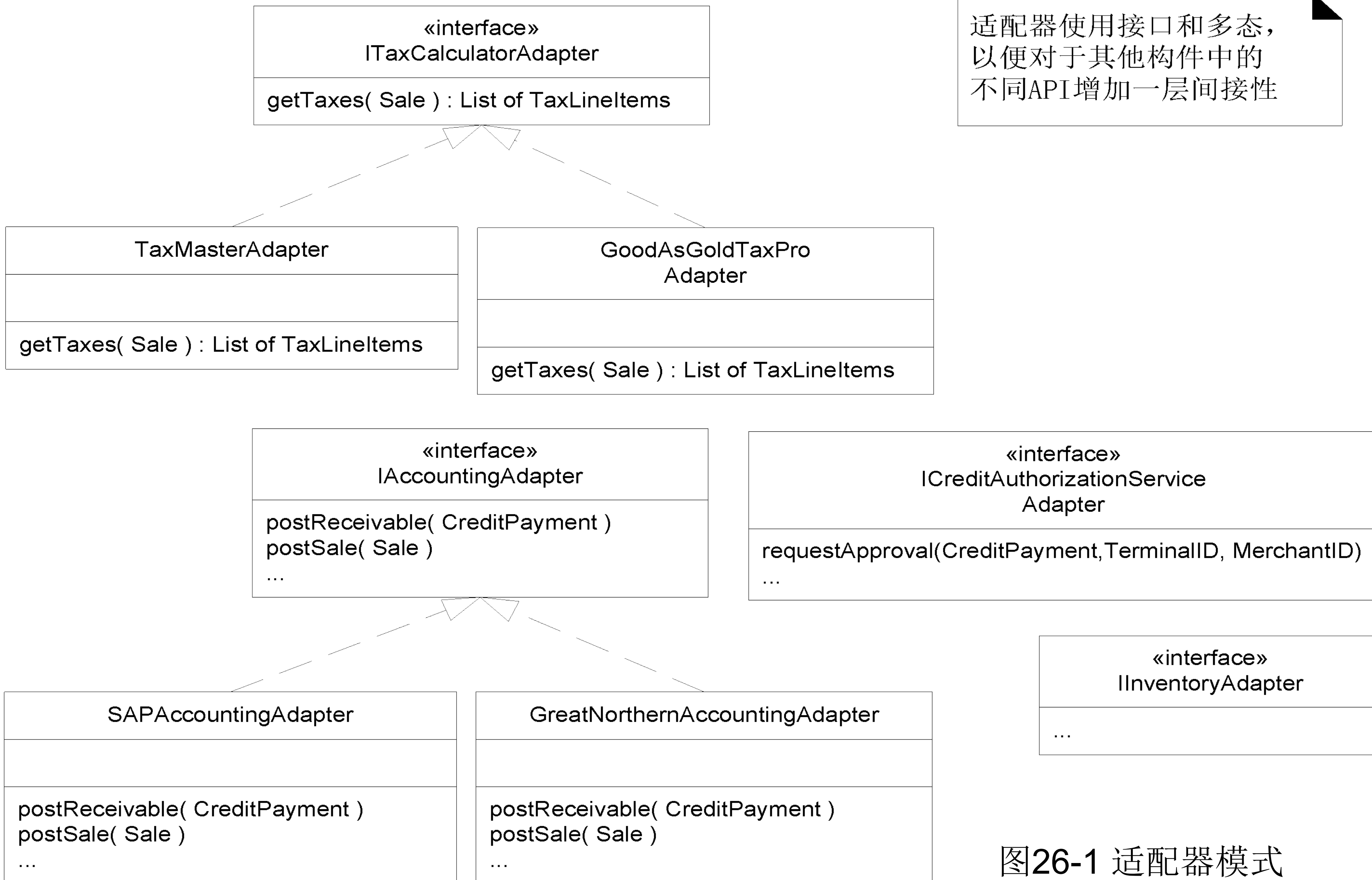


图26-1 适配器模式

使用适配器

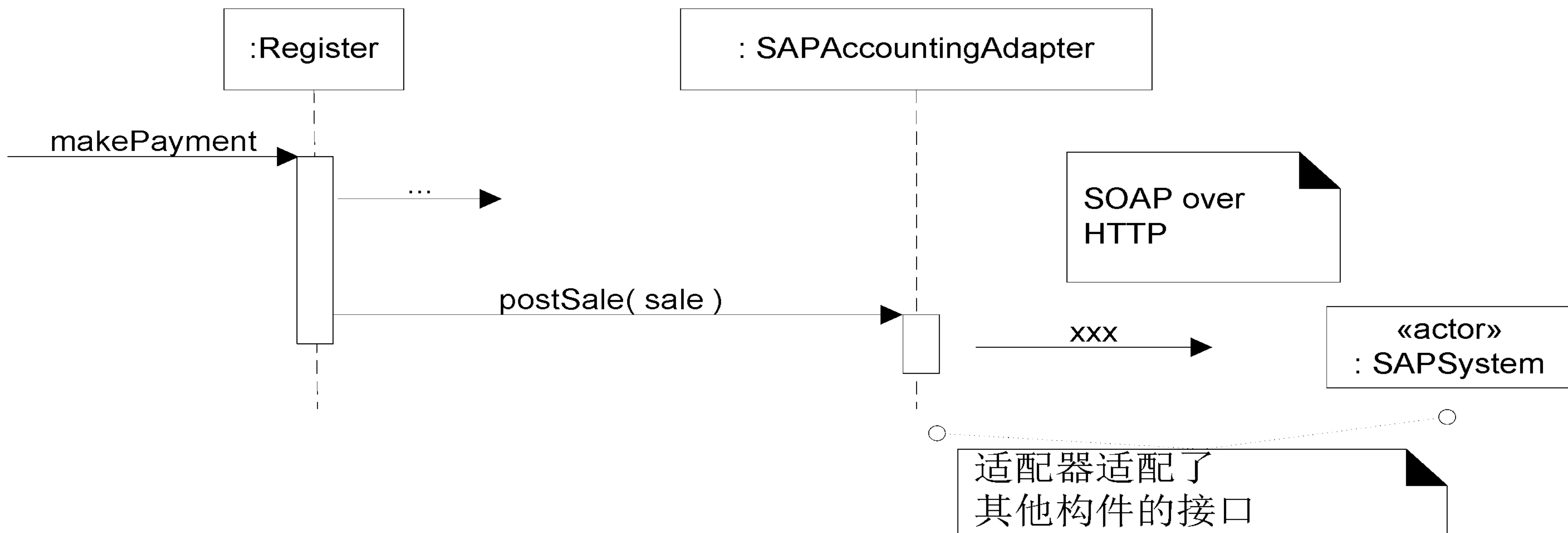


图26-2 使用适配器

准则：在类型名称中包含模式名称-易于理解使用了哪些模式

一些GRASP原则是对其他设计模式的归纳

- 适配模式可视为是某些GRASP构造模块的特化
  - 如：防止变异
- 问题：模式过多
  - Pattern Almanac 2000列出了大约500种设计模式，此后又发布了数百种
- 解决方案：找到根本原则
  - 详细了解和记住50种以上最重要的设计模式是非常重要的
  - 大多数设计模式可以被视为少数几个GRASP基本原则的特化，有助于加速学习和发现问题本质。

适配器与某些GRASP核心原则的关系

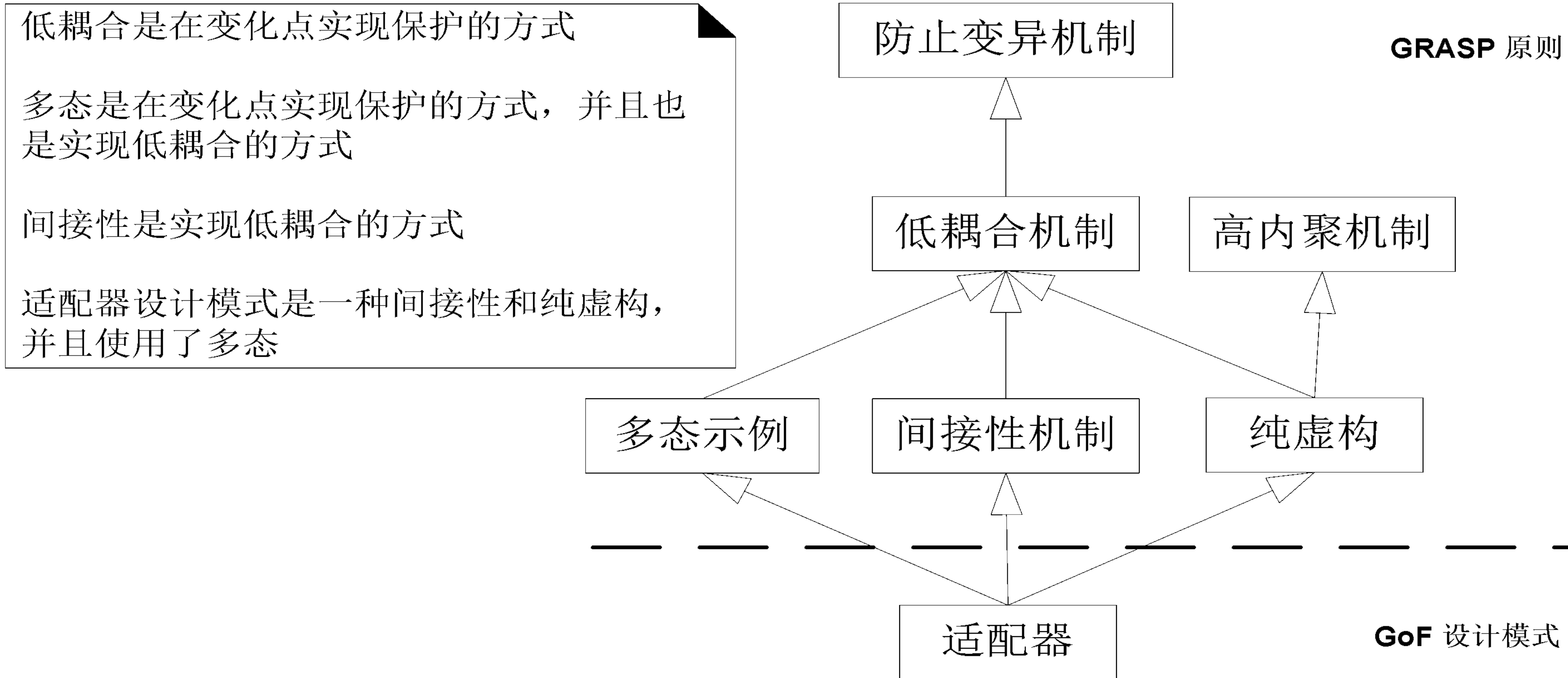


图26-3 适配器与某些GRASP核心原则的关系

在设计中发现的“分析”：领域模型

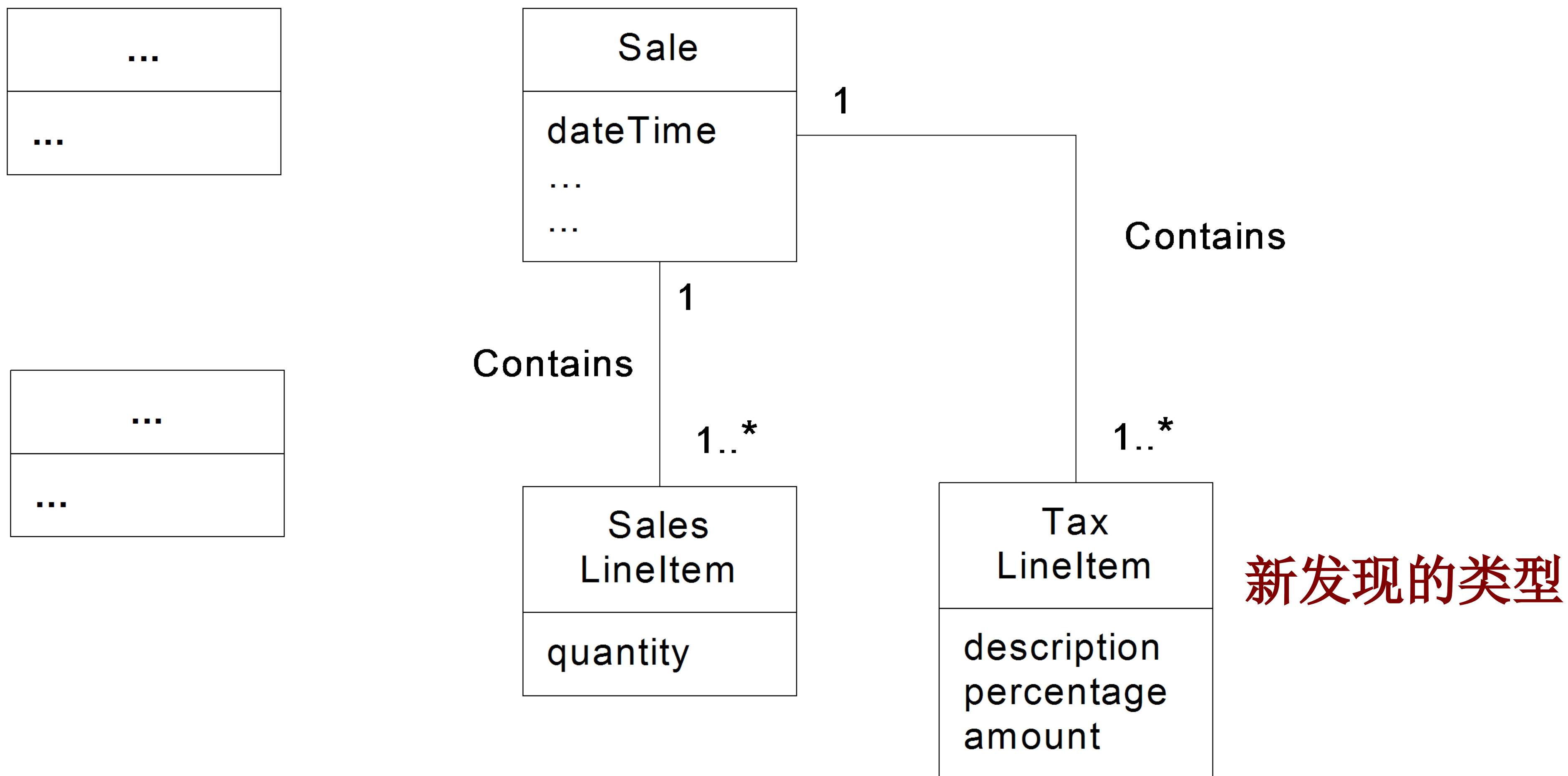


图26-4 更新后的部分领域模型

适配器模式引发了一个新的设计问题

由谁来创建Adapter呢？

如何确定创建哪个Adapter子类？例如，SAPAccountingAdapter还是 GreatNorthernAccountingAdapter

- 如果某些领域对象来创建Adapter,领域对象的职责将在考虑应用逻辑的同时，还要关注与外部服务构件的连通性，而决定这种连通性又具有复杂的创建逻辑。
- 按关注点分离的原则（separation of concerns），领域层的软件对象强调纯的应用逻辑职责，而关注与外部系统连通性的职责由另一组对象承担。

因此，我们考虑由一个专门的对象来创建Adapter。

工厂模式（Factory）

模式名：工厂（Factory）

问题：谁有责任创建一些特殊考虑的对象？比如说有复杂的创建逻辑，为了更好的内聚性而希望分离创建职责。

解决方案：创建一个称为工厂（Factory）的纯虚构对象来处理这种创建。

该模式也称为简单工厂（Simple Factory)或具体工厂（Concrete Factory）。是GoF抽象工厂（Abstract Factory）模式的简化。

工厂模式示例

ServicesFactory
accountingAdapter : IAccountingAdapter inventoryAdapter : IInventoryAdapter taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter getInventoryAdapter() : IInventoryAdapter getTaxCalculatorAdapter() : ITaxCalculatorAdapter ...

注意：工厂方法返回对象的类型是接口而不是类，因此工厂能够返回接口的任何实现

```
if ( taxCalculatorAdapter == null )
{
  //反射或数据驱动方法能够发现正确的类：从外部特性文件中读取外部特性

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

图26-5 工厂模式

工厂模式（Factory）引发的 新的设计问题

ServicesFactory 引起了一个新的设计问题：谁来创建Factory本身呢？如何调用它呢？

- 应注意到在处理过程中只需要ServicesFactory的一个实例。
- 编码中，ServicesFactory的方法需要在许多地方被调用（由于ServicesFactory创建相应的Adapter， Adapter调用外部服务系统）

于是，存在一个单一的可见性问题：

如何获得对这个单一的ServicesFactory实例的可见性?

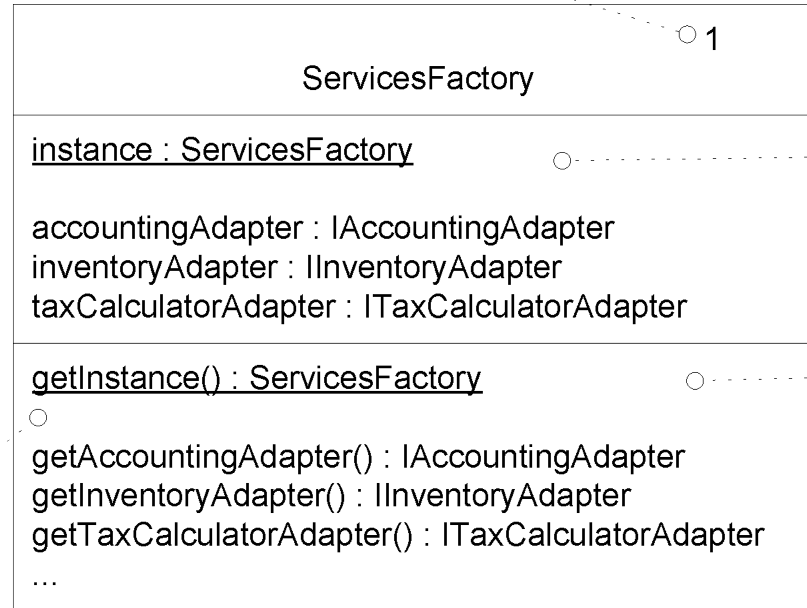
单实例类模式（Singleton）

模式名：单实例类（Singleton）

问题：如何使一个类严格地只有一个实例？

解决方案：对类定义静态方法以返回单实例。

ServiceFactory类中的单实例类模式




```
// 静态方法
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
  instance = new ServicesFactory()
return instance
}
```

图26-6 在ServiceFactory类中的单实例类模式

单实例类的顺序图

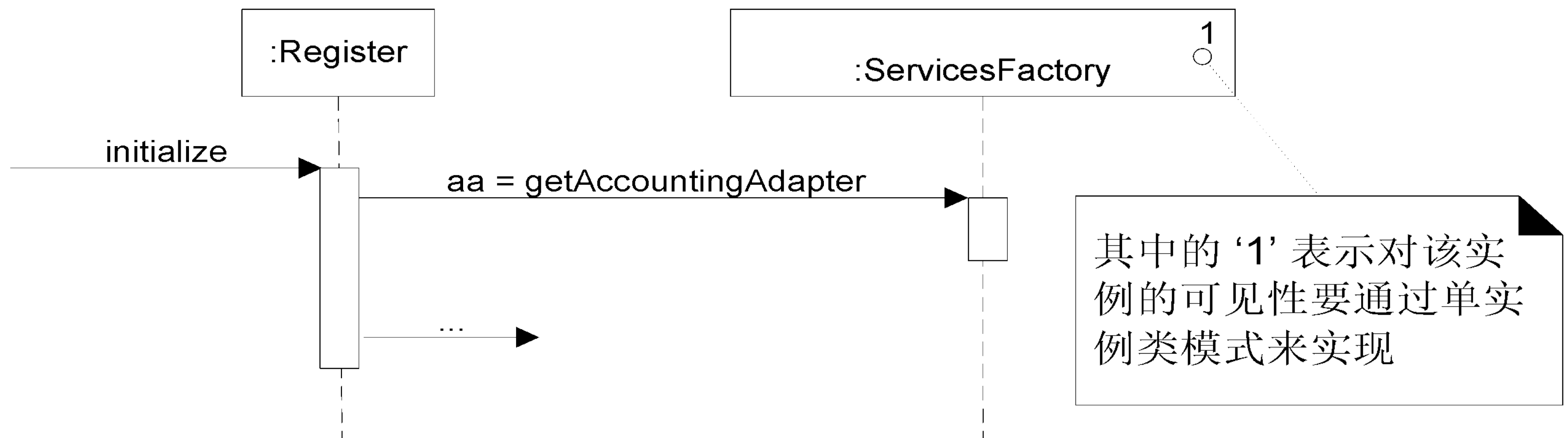


图26-7 在UML中含有标记“1”暗示了getAccountingAdapter是单实例类模式的消息

独身模式（singleton）代码示例

```
{
// static method of the ServicesFactory
public static synchronized ServicesFactory getInstance()
{
  if ( instance == null )
    instance = new ServicesFactory()
  return instance
}
}
```

```
public class Register {
public void initialize()
{
  ... do some work ...
  // accessing the singleton Factory via the getInstance call
  accountingAdapter =
      ServicesFactory.getInstance().getAccountingAdapter();
  ... do some work ... }
// other methods... }
```

通过Singleton的Factory按创建逻辑创建Adapter的实例

具有变化接口的外部服务系统问题的结论

为了使系统满足适应外部服务系统的可变接口，我们应用**适配器模式**，设计具有同一接口的Adapters。

为了按系统的要求灵活选择所需的Adapter，我们应用**工厂模式**，设计一个Factory。 Factory按系统当前的要求实例化相应的Adapter。

为了系统中同一类的Adapters实例具有唯一的创建（选择）逻辑，我们应用**单实例类**模式，使Factory在系统中仅有唯一的一个实例。

适配器、工厂和单实例类模式应用于设计

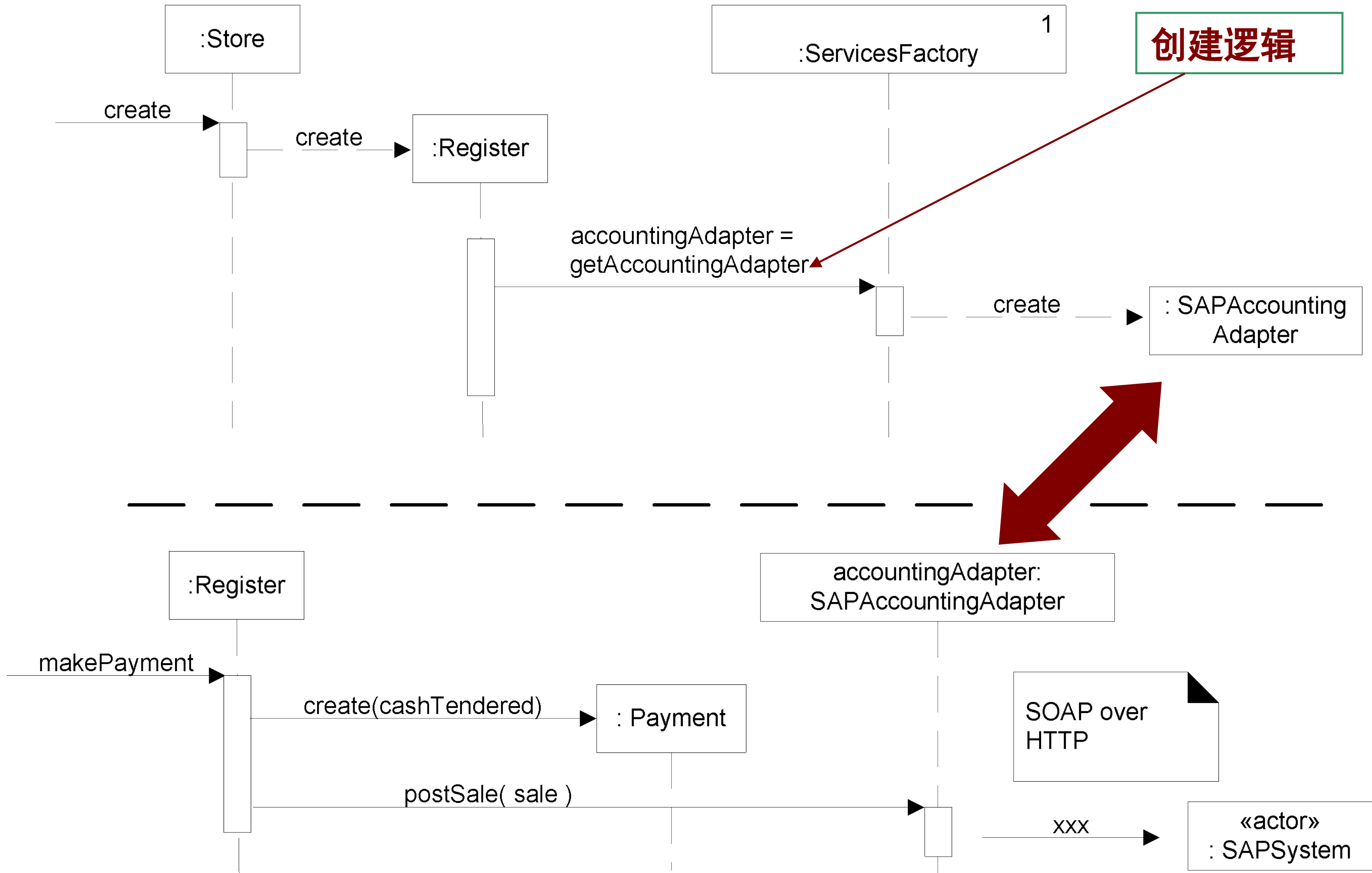


图26-8 适配器、工厂和单实例类模式应用于设计

应用设计模式解决支持复杂的定价规则问题

定价策略1：PercentDiscount

- 所有商品都打一个百分比折扣。

定价策略2：AbsoluteDiscountOverThreshold

- 当一次销售总金额大于一个定额时，减去一个绝对数的金额。

……

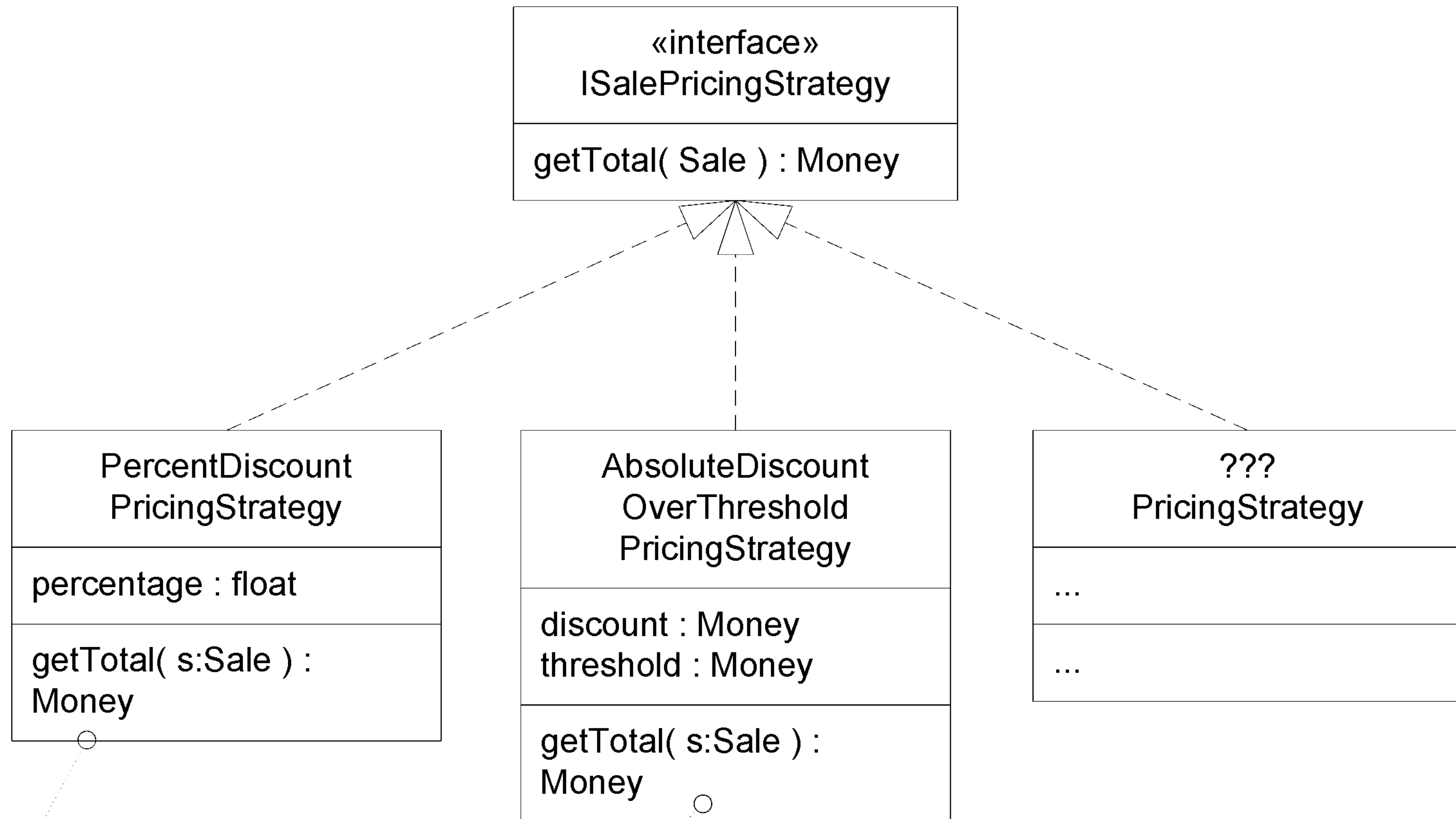
策略模式（Strategy）

模式名：策略（strategy）

问题：如何设计变化但相关的算法或政策？如何设计才能使这些算法或政策具有可变更的能力？

解决方案（建议）：在单独的类中分别定义每种算法/政策/策略，并且使其具有共同接口

定价策略类

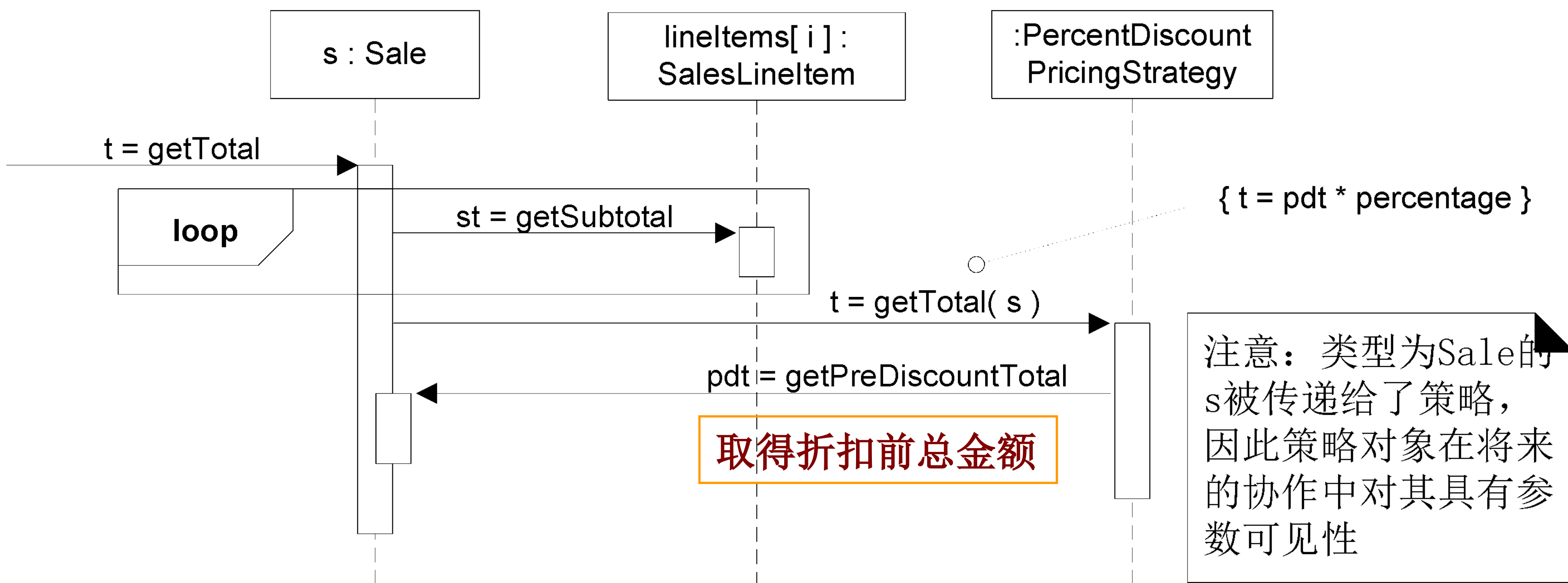



```
{
  return s.getPreDiscountTotal() * percentage
}
```

```
{
pdt := s.getPreDiscountTotal()
if ( pdt < threshold )
   return pdt
else
   return pdt - discount
}
```

图26-9 定价策略类

协作中的策略



当消息getTotal发给Sale时，它就委派一些工作给它的策略对象，通常，语境对象将自己传给策略对象，以便策略对象对语境对象有一个可见性（可以协作完成委派的工作）。

图26-10 协作中的策略

语境对象需要其策略的属性可见性

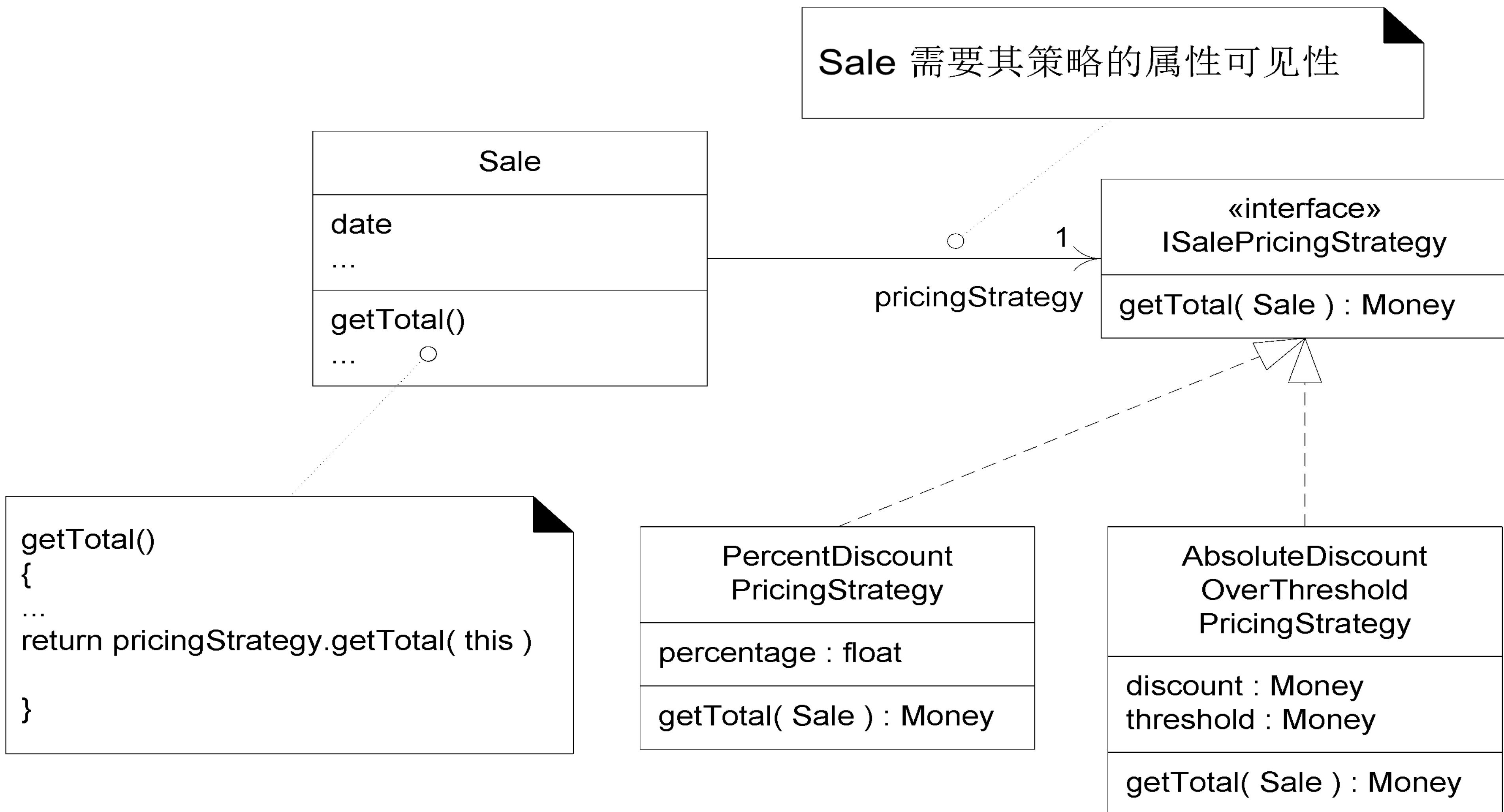


图26-11 语境对象需要其策略的属性可见性

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/165142123204011320