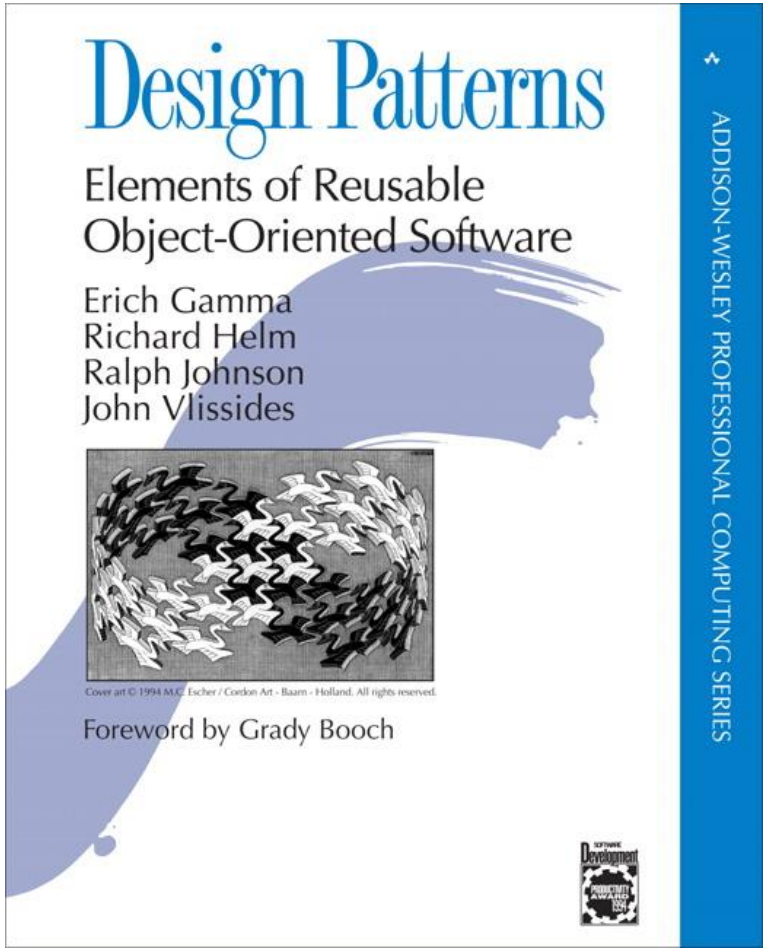
Applying Gang-of-Four Design Patterns

Dalibor Dvorski (ddvorski@conestogac.on.ca)
School of Engineering and Information Technology
Conestoga College Institute of Technology and Advanced Learning

The Gang-of-Four (GoF)



"The authors of the Design Patterns book came to be known as the 'Gang-of-Four'. The name of the book is too long for e-mail, so 'book by the gang of four' became a shorthand name for it. After all, it isn't the only book on patterns. That got shortened to 'GoF book', which is pretty cryptic the first time you hear it."
(Cunningham & Cunningham, Inc.)

Adapter

Problem: _____

_____

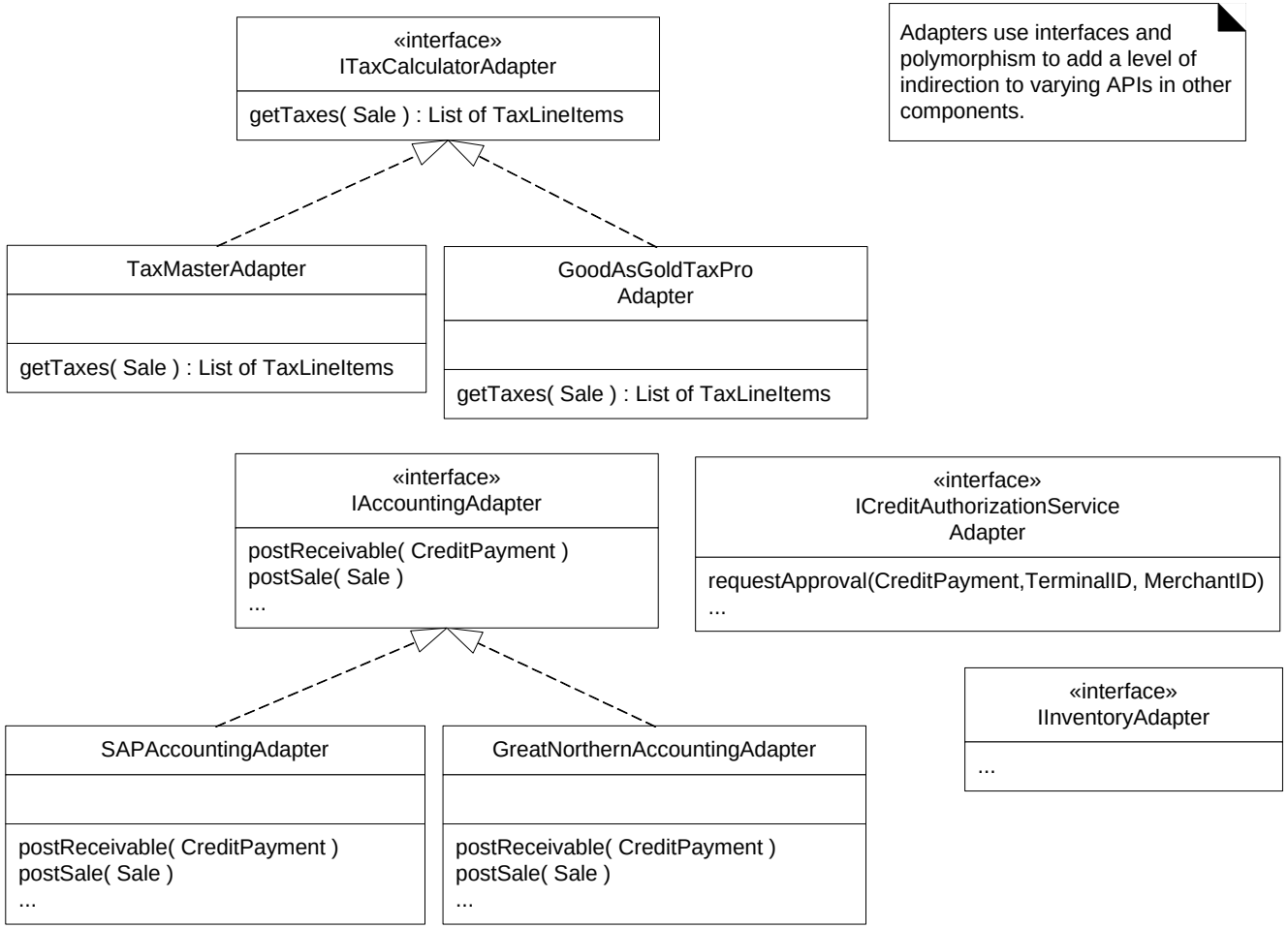
Solution: _____

_____

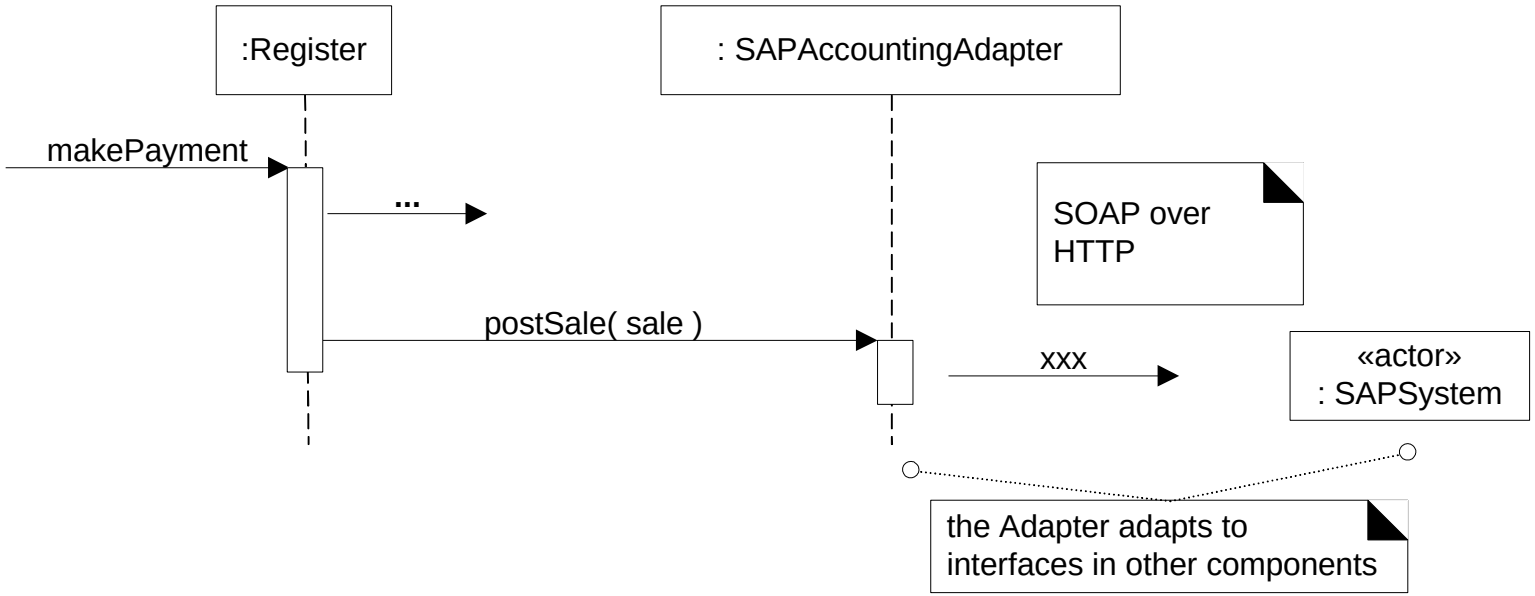
Adapter supports Protected Variations with respect to changing external interfaces or third-party packages through the use of an Indirection object that applies interfaces and Polymorphism.

e.g. The NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed. A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application. A particular adapter instance will be instantiated for the chosen external service, such as SAP for accounting, and will adapt the `postSale` request to the external interface.

Adapter


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
«interface»
IAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
ICreditAuthorizationService Adapter
requestApproval(CreditPayment,TerminalID, MerchantID)
...
«interface»
IInventoryAdapter
...
SAPAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
GreatNorthernAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...

Adapter


:Register
: SAPAccountingAdapter
makePayment
...
SOAP over HTTP
postSale(sale)
xxx
«actor»
: SAPSystem
the Adapter adapts to interfaces in other components

Pattern Overload!

Hundreds of design patterns have been published. The curious developer has no time to actually program given this reading list! Yes, it's important for a designer to know the most important design patterns, but few of us can learn or remember so many patterns, or even start to organize that pattern plethora into a useful taxonomy. But there's good news: most design patterns can be seen as specializations of a few basic GRASP principles. Although it is indeed helpful to study detailed design patterns to accelerate learning, it is even more helpful to see their underlying basic themes to help us cut through the myriad details and see the essential "alphabet" of design techniques being applied.

"One comment I saw … someone claiming that in a particular program they tried to use all 23 GoF patterns. They said they had failed, because they were only able to use 20. They hoped the client would call them again to come back again so maybe they could squeeze in the other 3 … Trying to use all the patterns is a bad thing, because you will end up with synthetic designs – speculative designs that have flexibility that no one needs. These days software is too complex. We can't afford to speculate what else it should do. We need to really focus on what it needs. That's why I like refactoring to patterns. People should learn that when they have a particular kind of problem or code smell, as people call it these days, they can go to their patterns toolbox to find a solution."
(Erich Gamma)

Factory

Problem: _____

_____

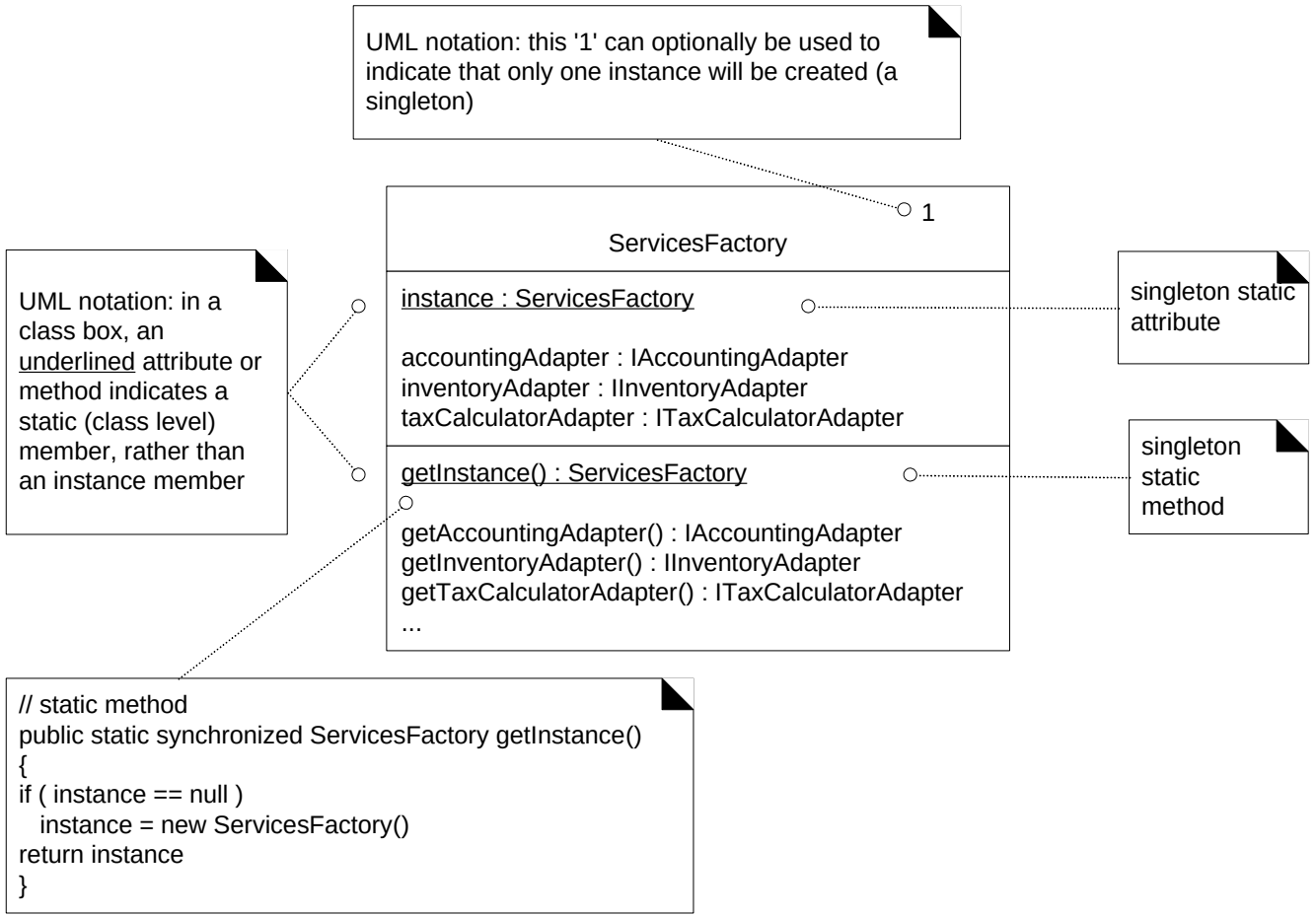
_____

Solution: Create a _____ object called a Factory that handles the creation.

Not a GoF pattern, but extremely widespread. Also called *Simple Factory* or *Concrete Factory*.

e.g. In the prior Adapter pattern solution for external services with varying interfaces, who creates the adapters? And how to determine which class of adapter to create, such as `TaxMasterAdapter` or `GoodAsGoldTaxProAdapter`? If some domain object creates them, the responsibilities of the domain object are going beyond pure application logic (such as sales total calculations) and into other concerns related to connectivity with external software components. (Recall previous discussion on separation of concerns, modularization, layering, etc.) Therefore, choosing a domain object (such as `Register`) to create the adapters does not support the goal of a separation of concerns, and lowers its cohesion.

Factory


UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)
1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
singleton static attribute
singleton static method
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
// static method
public static synchronized ServicesFactory getInstance()
{
if (instance == null)
instance = new ServicesFactory()
return instance
}

Strategy

Problem: _____

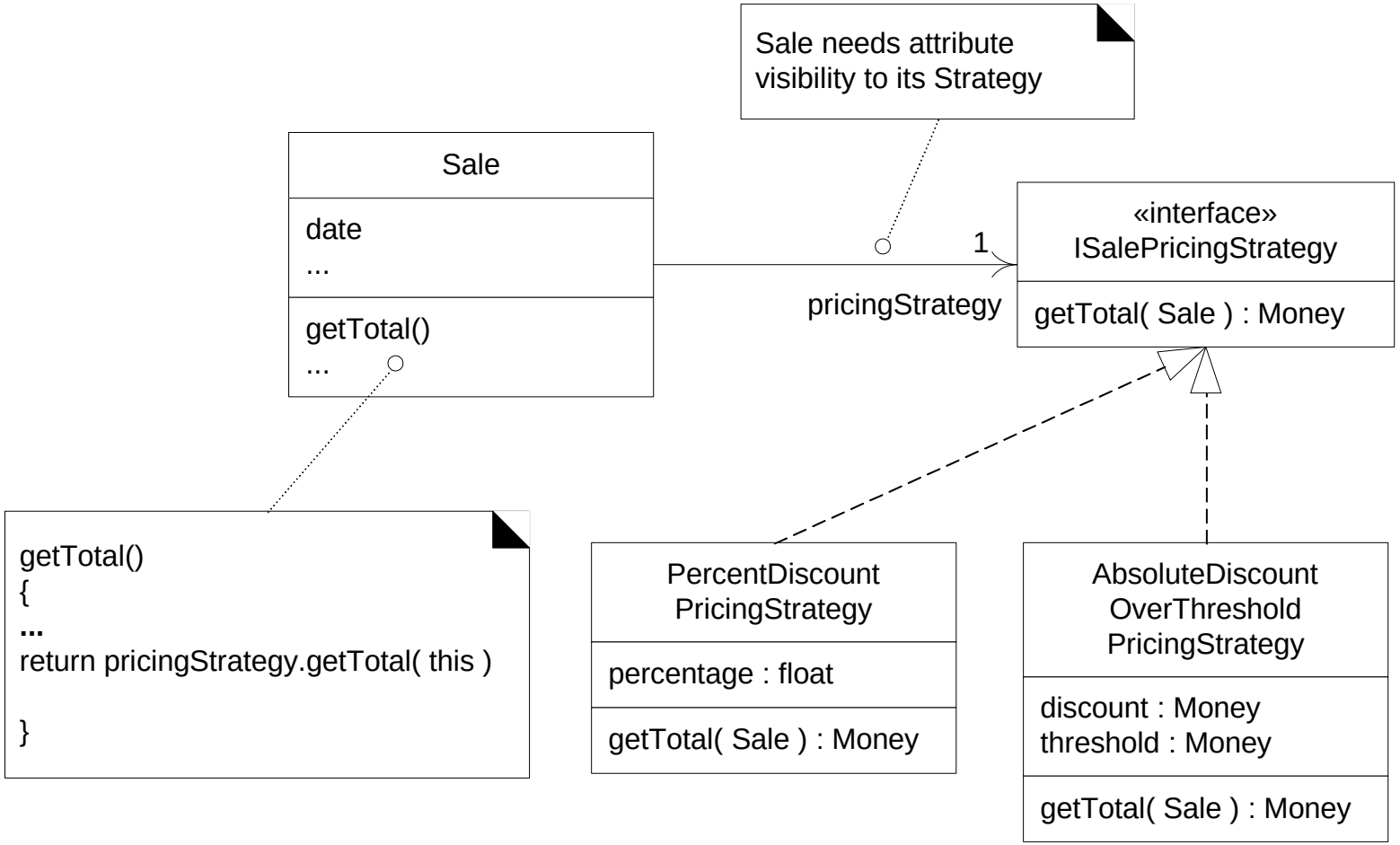
_____

Solution: _____

_____

e.g. The pricing strategy (which may also be called a rule, policy, or algorithm) for a store can vary. How do we design for these varying pricing algorithms? Since the behaviour of pricing varies by the strategy (or algorithm), we create multiple `SalePricingStrategy` classes, each with a polymorphic `getTotal` method. Each `getTotal` method takes the `Sale` object as a parameter, so that the pricing strategy object can find the pre-discount price from the `Sale`, and then apply the discounting rule. The implementation of each `getTotal` method will be different: `PercentDiscountPricingStrategy` will discount by a percentage, and so on.

Strategy


Sale needs attribute visibility to its Strategy
Sale
date
...
getTotal()
...
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
getTotal()
{
...
return pricingStrategy.getTotal(this)
}
PercentDiscount
PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount
OverThreshold
PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money

Strategy with Factory

Who should create the strategy? _____

Composite

Problem: _____

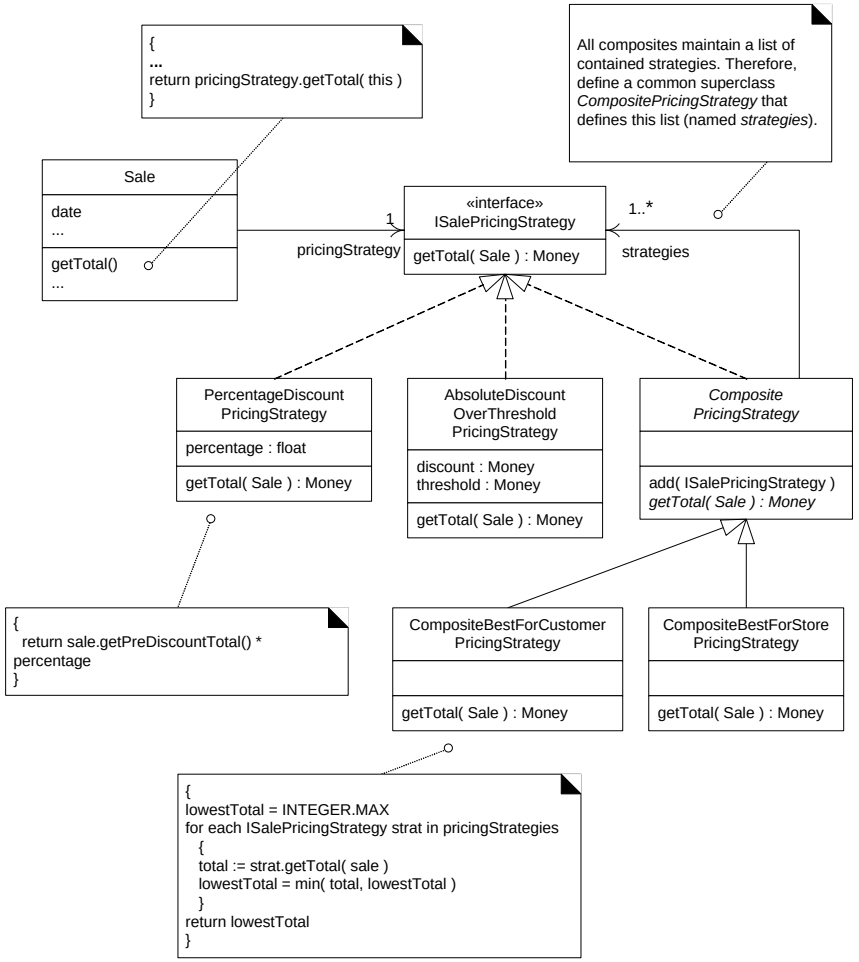
_____

Solution: _____

_____

e.g. How do we handle the case of multiple, conflicting pricing policies? Usually, a store applies the “best for the customer” (lowest price) conflict resolution strategy, but this is not required, and it could change. Is there a way to change the design so that the `Sale` object does not know if it is dealing with one or many pricing strategies, and also offer a design for the conflict resolution? A new class called `CompositeBestForCustomerPricingStrategy` can implement the `ISalesPricingStrategy` interface and itself contain other `ISalesPricingStrategy` objects.

Composite



Observe that in this design, the composite classes such as `CompositeBestForCustomerPricingStrategy` inherit an attribute `pricingStrategies` that contains a list of more `ISalePricingStrategy` objects. This is a signature feature of a composite object: the outer composite object contains a list of inner objects, and both the outer and inner objects implement the same interface. That is, the composite class itself implements the `ISalePricingStrategy` interface. Thus, we can attach either a composite `CompositeBestForCustomerPricingStrategy` object or an atomic `PercentDiscountPricingStrategy` object to the `Sale` object, and the `Sale` does not know or care if its pricing strategy is an atomic or composite strategy – it looks the same to the `Sale` object. It is just another object that implements the `ISalePricingStrategy` interface and understands the `getTotal` message.

Façade

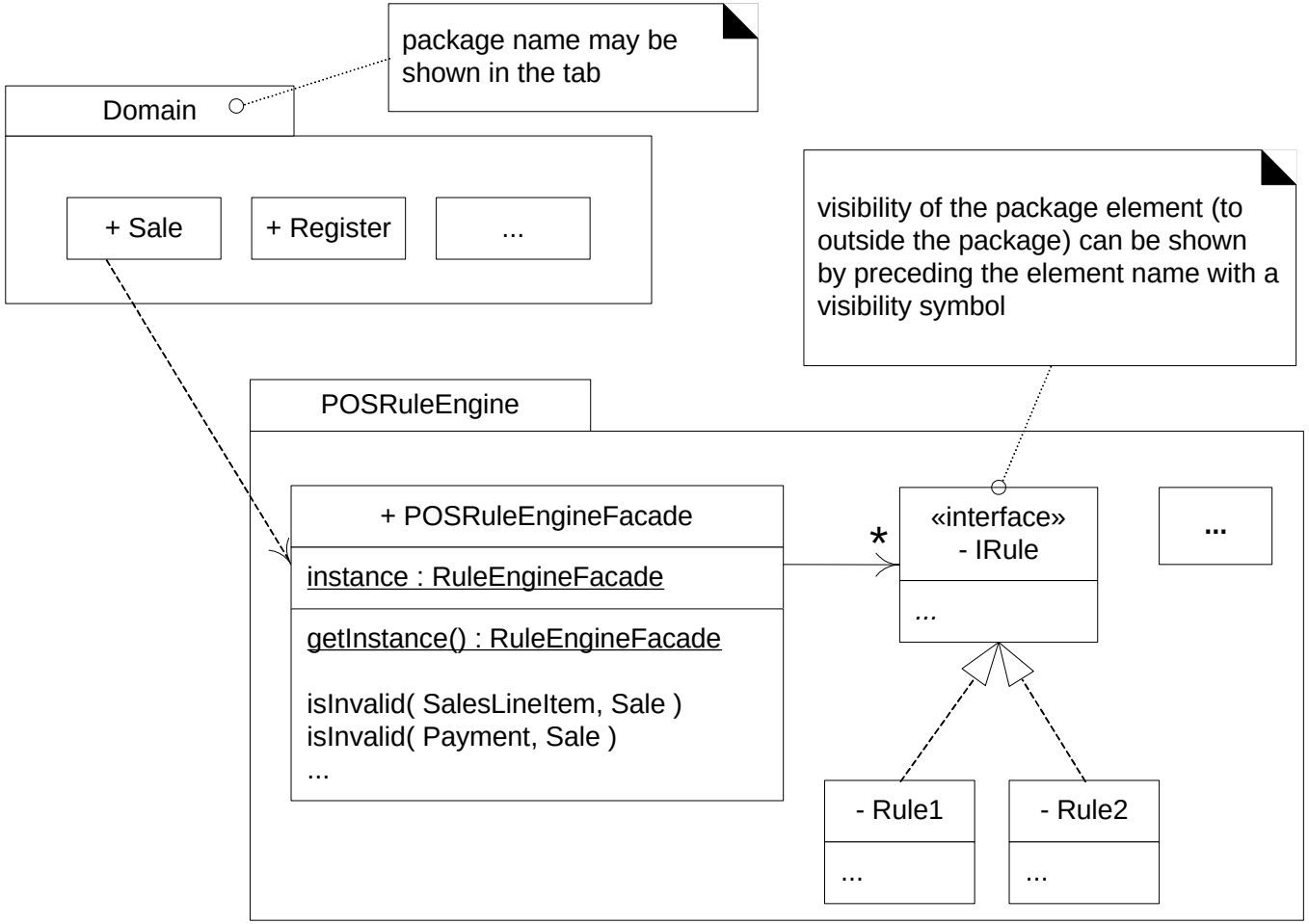
Problem: _____
_____
_____

Solution: _____
_____
_____

_____
_____
_____

e.g. Suppose that the software architect wants a design that has low impact on the existing software components. That is, she wants to design for a separation of concerns, and factor out this rule handling into a separate concern. Furthermore, suppose that the architect is unsure of the best implementation for this pluggable rule handling, and may want to experiment with different solutions for representing, loading, and evaluating the rules. For instance, rules can be implemented with the Strategy pattern, or with free open-source rule interpreters that read and interpret a set of if-then rules, or with commercial, purchased rule interpreters, among other solutions.

Façade



e.g. We will define a “rule engine” subsystem, whose specific implementation is not yet known. It will be responsible for evaluating a set of rules against an operation (by some hidden implementation), and then indicating if any of the rules invalidated the operation. The façade object to this subsystem will be called `POSRuleEngineFacade`.

Observer

Problem:

Solution:

e.g.

Assigned Readings

1. Chapter 26: Applying GoF Design Patterns from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
   - Read §26.5 for the Singleton pattern, while also reviewing notes from last term.
   - Pay attention to the additional examples and design explanations.
2. The observer design pattern from http://dofactory.com/net/observer-design-pattern/ and http://sourcemaking.com/design_patterns/observer/ and
   Chapter 2 : The Observer Pattern: Keeping your Objects in the Know from
   Head First Design Patterns.
   - Implement the observer design pattern using C#, Java, and PHP.
   - Future exercises and assignments will make use of this pattern.