Construcción de Software

Capítulo 3

Patrones de diseño

Contenidos

1. Concepto de patrón
2. Clasificación de patrones
3. Estudio de algunos de los principales patrones GoF
    - Adaptador
    - Factoría
    - Singleton
    - Estrategia
    - Composite
    - Fachada
    - Observador

Bibliografía

- [Larman 02] Larman, C. "*UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado*", Segunda Edición, Prentice-Hall, 2002. Capítulo 23.
- [Larman 05] Larman, C. "*Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development*", 3rd edition, Prentice-Hall, 2005. Capítulo 26.
- [Gamma et al. 02] Gamma E., et al., *Patrones de Diseño*, Addison-Wesley, 2002.

Arquitectura software y patrones

"Una arquitectura orientada a objetos bien estructurada está llena de patrones. La calidad de un sistema orientado a objetos se mide por la atención que los diseñadores han prestado a las colaboraciones entre sus objetos."

"Los patrones conducen a arquitecturas más pequeñas, más simples y más comprensibles"

G. Booch

Diseño orientado a objetos

- “Diseñar software orientado a objetos es difícil pero diseñar software orientado a objetos reutilizable es más difícil todavía. Diseños generales y flexibles son muy difíciles de encontrar la primera vez”

- ¿Qué conoce un programador experto que desconoce uno inexperto?

Reutilizar soluciones que funcionaron en el pasado: EXPERIENCIA

Patrones

- Describen un problema recurrente y una solución.
- Cada patrón nombra, explica, evalúa un diseño recurrente en sistemas OO.
- Elementos principales:
  - **Nombre**
  - **Problema**
  - **Solución:** Descripción abstracta
  - **Consecuencias**

Granularidad

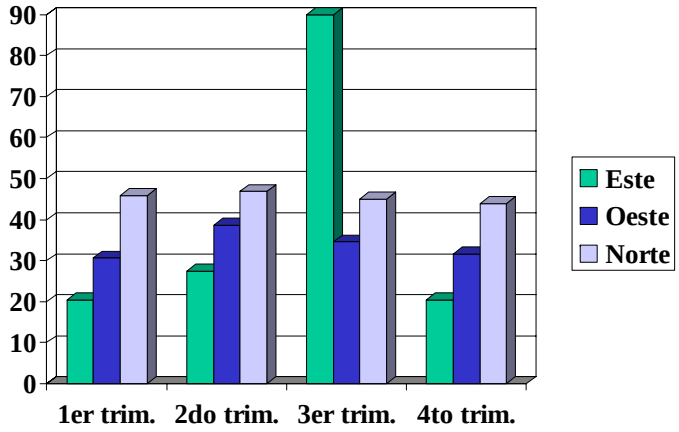
- ***Patrones de Código***
  - Nivel de lenguaje de programación
- ***Frameworks***
  - Diseños específicos de un dominio de aplicaciones.
- ***Patrones de Diseño***
  - Descripciones de clases cuyas instancias colaboran entre sí que deben ser adaptados para resolver problemas de diseño generales en un contexto particular.
  - Un patrón de diseño identifica: ***Clases***, ***Instancias***, ***Roles, Colaboraciones*** y la ***Distribución de responsabilidades***.

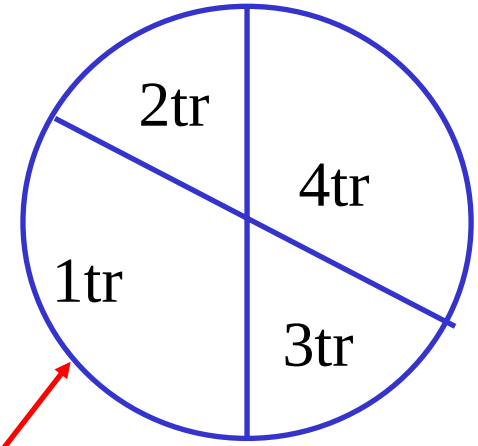
Patrones y *frameworks*

- Un *framework* es una colección organizada de clases que constituyen un diseño reutilizable para un dominio específico de software.
- Necesario adaptarlo a una aplicación particular.
- Establece la arquitectura de la aplicación
- Diferencias:
  - Patrones son más abstractos que los *frameworks*
  - Patrones son elementos arquitecturales más pequeños
  - Patrones son menos especializados

Modelo-Vista-Control
(*MVC paradigm* o *MVC framework*)

- Utilizado para construir interfaces de usuario en Smalltalk-80.
- Basado en tres tipos de objetos:
  - **Modelo**: objetos del dominio
  - **Vista**: objetos presentación en pantalla (interfaz de usuario)
  - **Controlador**: define la forma en que la interfaz reacciona a la entrada del usuario.
- Desacopla el modelo de las vistas.





Vistas

Modelo

t1=20
t2=25
t3=90
t4=20

	1tr.	2tr.	3tr.	4tr.
Este	20	25	90	20
Oeste	30	38	32	32
Norte	47	47	45	45

Modelo-Vista-Control

- Utiliza los siguientes patrones:
  - ***Observer***
  - ***Composite***
  - ***Strategy***
  - ***Decorator***
  - ***Factory method***

Plantilla de definición

- **Nombre del patrón**
- **Propósito**
  - ¿Qué hace? ¿Cuál es su razón y propósito? ¿Qué cuestión de diseño particular o problema aborda?
- **Sinónimos**
- **Motivación**
  - Escenario que ilustra un problema particular y cómo el patrón lo resuelve.

Plantilla de definición

- **Aplicabilidad**
  - ¿En qué situaciones puede aplicarse? ¿Cómo las reconoces? Ejemplos de diseños que pueden mejorarse
- **Estructura**
  - Diagramas de clases que ilustran la estructura
- **Participantes**
  - Clases que participan y sus responsabilidades
- **Colaboraciones**
  - Diagramas de interacción que muestran cómo colaboran los participantes.

Plantilla de definición

- **Consecuencias**
  - ¿Cómo alcanza el patrón sus objetivos? ¿Cuáles son los compromisos y resultados de usar el patrón? Alternativas, Costes y Beneficios
- **Implementación**
  - Técnicas, heurísticas y consejos para la implementación
  - ¿Hay cuestiones dependientes del lenguaje?
- **Ejemplo de código**
- **Usos conocidos**
- **Patrones relacionados**

Clasificación de patrones de diseño

		Propósito		
		Creación	*Estructural*	*Comportamiento*
Ámbito	*Clase*	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Adaptador

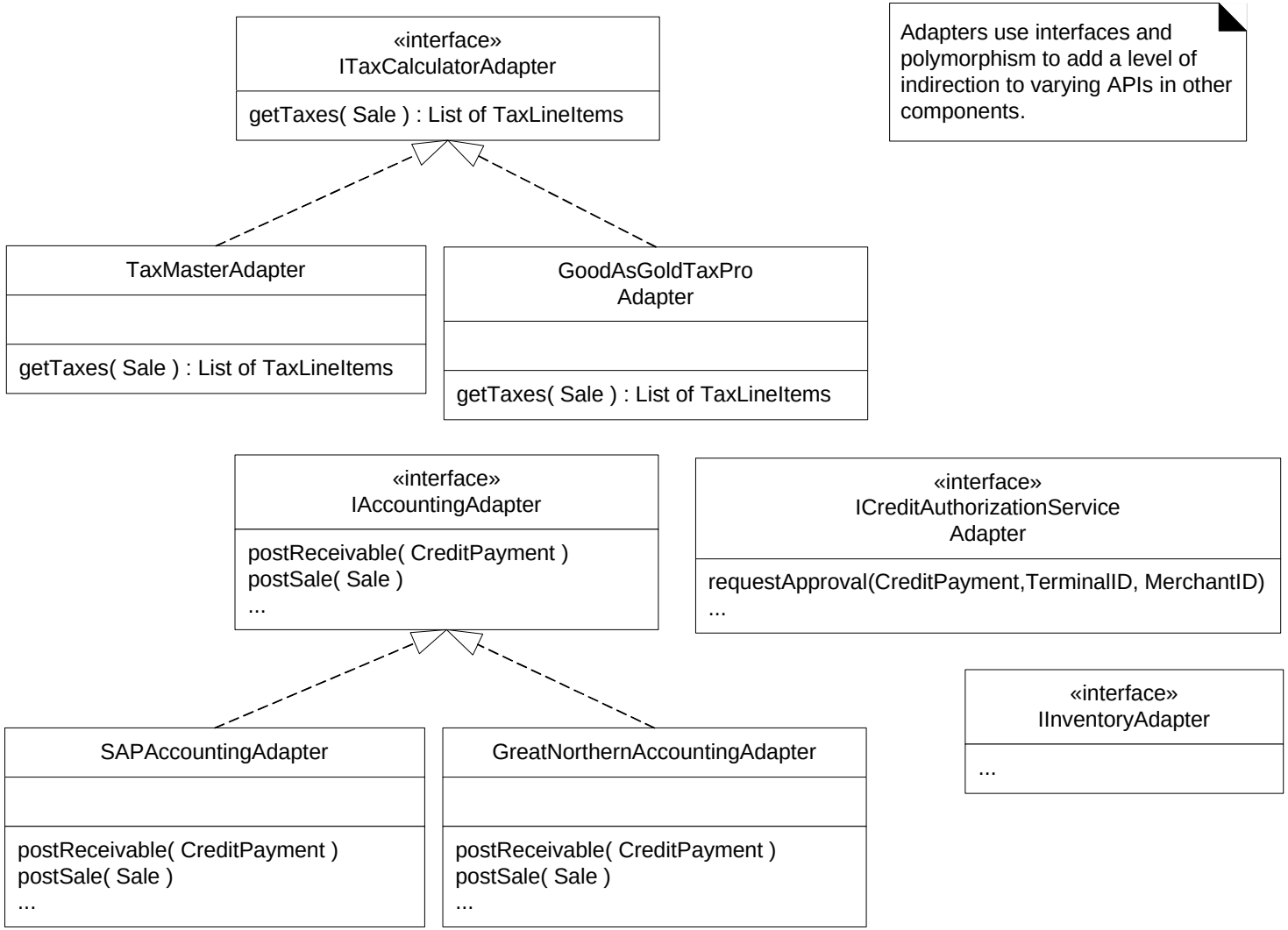
En *NuevaEra*...

- La aplicación *NuevaEra* necesita soportar diferentes tipos de servicios externos de terceras partes, entre los que se encuentran los calculadores de impuestos, servicios de autorización de pagos, sistemas de inventario, y sistemas de contabilidad. Cada uno tiene una API diferente, que no puede ser cambiada.
- Una solución es añadir un nivel de indirección con objetos que adapten las distintas interfaces externas a una interfaz consistente que es la que se utiliza en la aplicación.

Adaptador (GoF)

- Nombre: Adaptador (*Adapter / Wrapper*)
- Contexto/Problema: ¿Cómo resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes similares con diferentes interfaces?
- Solución (consejo): Convertir la interfaz original de un componente en otra interfaz, a través de un objeto *adaptador* intermedio.

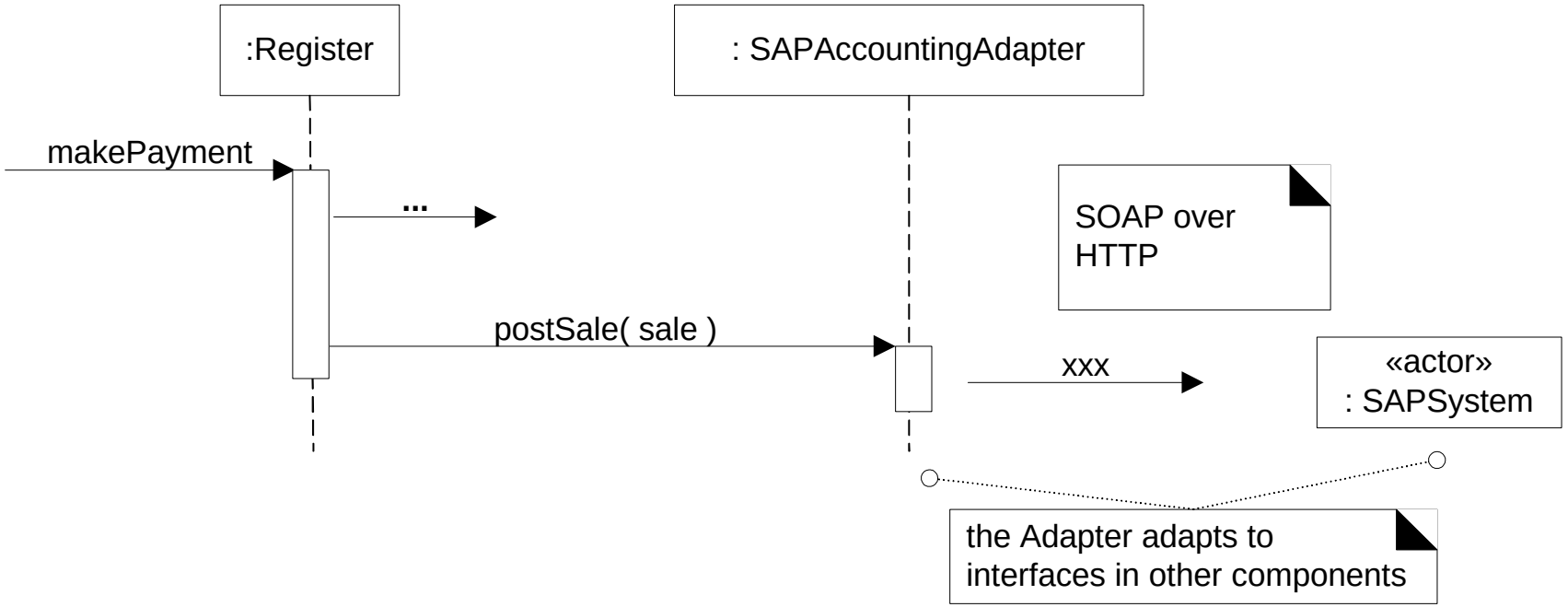
Adaptador


«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)
...

Adaptador

- Una instancia de adaptador será instanciada para el servicio externo elegido, tal como *SAPAccountingAdapter*, y adaptará la petición *postSale* al interfaz externo, como p.ej. un interfaz SOAP XML sobre HTTPS para un servicio Web de intranet ofrecido por SAP.
- *Guía*: Es común es que el nombre del patrón se inserte en los nombres de los tipos involucrados, de forma que se transmita rápidamente qué patrones están involucrados en un fragmento de código.

Adaptador


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

Factoría

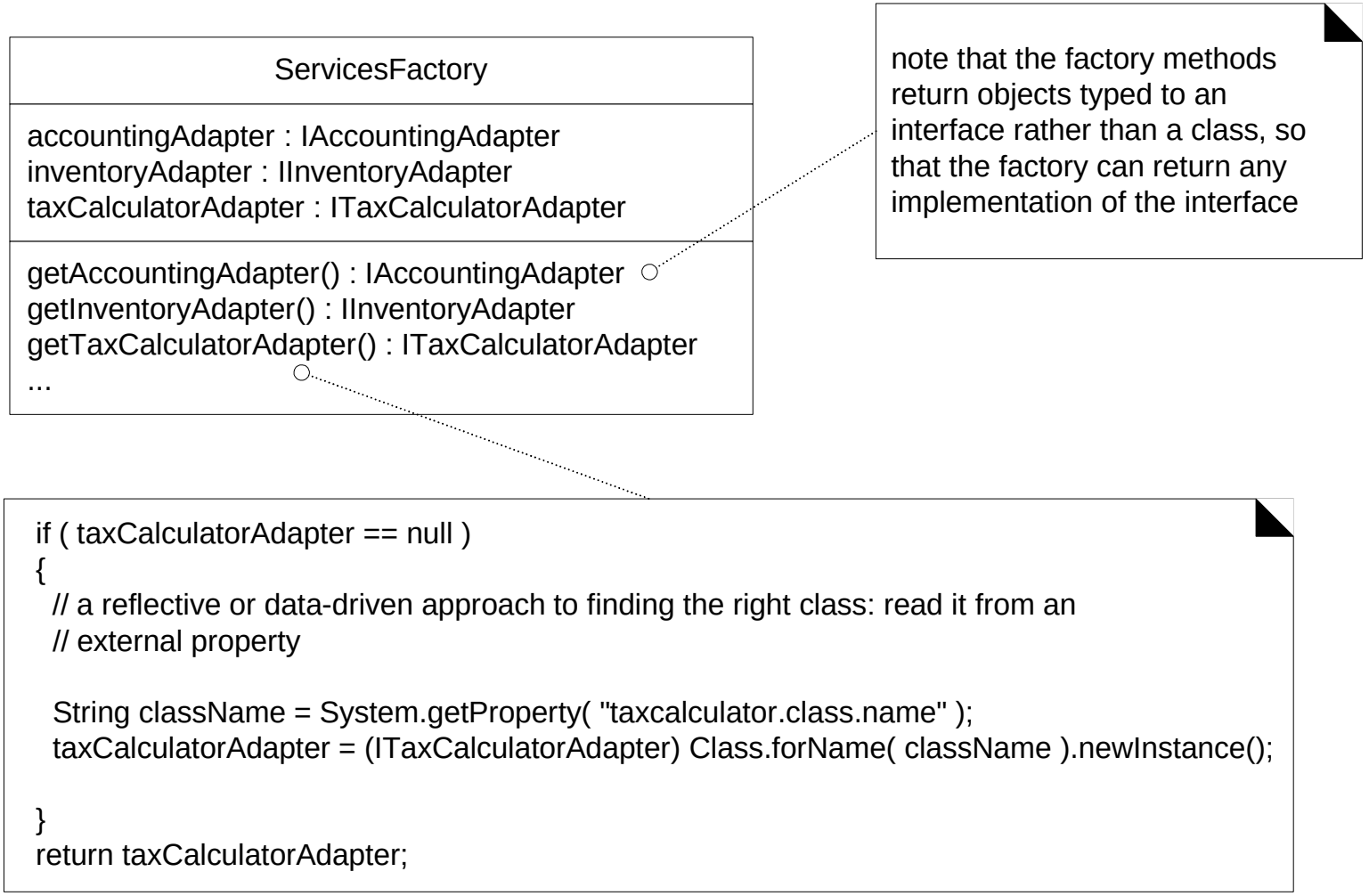
En *NuevaEra*...

- ¿Quién crea el adaptador? ¿Y cómo determinar qué clase de adaptador crear?
- Si los creara algún objeto del dominio, las responsabilidades de los objetos del dominio excederían la lógica pura de la aplicación y entrarían en cuestiones relacionadas con la conexión con componentes software externos.

Factoría

- Nombre: Factoría concreta (*Simple Factory / Concrete Factory)*
- Contexto/Problema: ¿Quién debe ser el responsable de la creación de los objetos cuando existen consideraciones especiales, como una lógica de creación compleja, el deseo de separar las responsabilidades de la creación para mejorar la cohesión, etc.?
- Solución (consejo): Crear un objeto *Fabricación Pura* (es decir, que no pertenece al modelo conceptual) denominado *Factoría* que maneje la creación.

Factoría


ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface
if (taxCalculatorAdapter == null)
{
// a reflective or data-driven approach to finding the right class: read it from an
// external property
String className = System.getProperty("taxcalculator.class.name");
taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName(className).newInstance();
}
return taxCalculatorAdapter;

Factoría

Los objetos factoría tienen varias ventajas:

- Separan la responsabilidad de la creación compleja en objetos de apoyo cohesivos.
- Ocultan la lógica de la creación potencialmente compleja.
- Permiten introducir estrategias para mejorar el rendimiento de la gestión de la memoria, como objetos caché o de reciclaje.

Factoría

En *NuevaEra*…

- La lógica para decidir qué clase se crea se resuelve leyendo el nombre de la clase de una fuente externa (p.ej., por medio de una propiedad del sistema, si se usa Java) y después se carga la clase dinámicamente.
- Es un ejemplo de *diseño dirigido por los datos*.
- A menudo se accede a las factorías con el patrón *Singleton*.

Singleton

En *NuevaEra*...

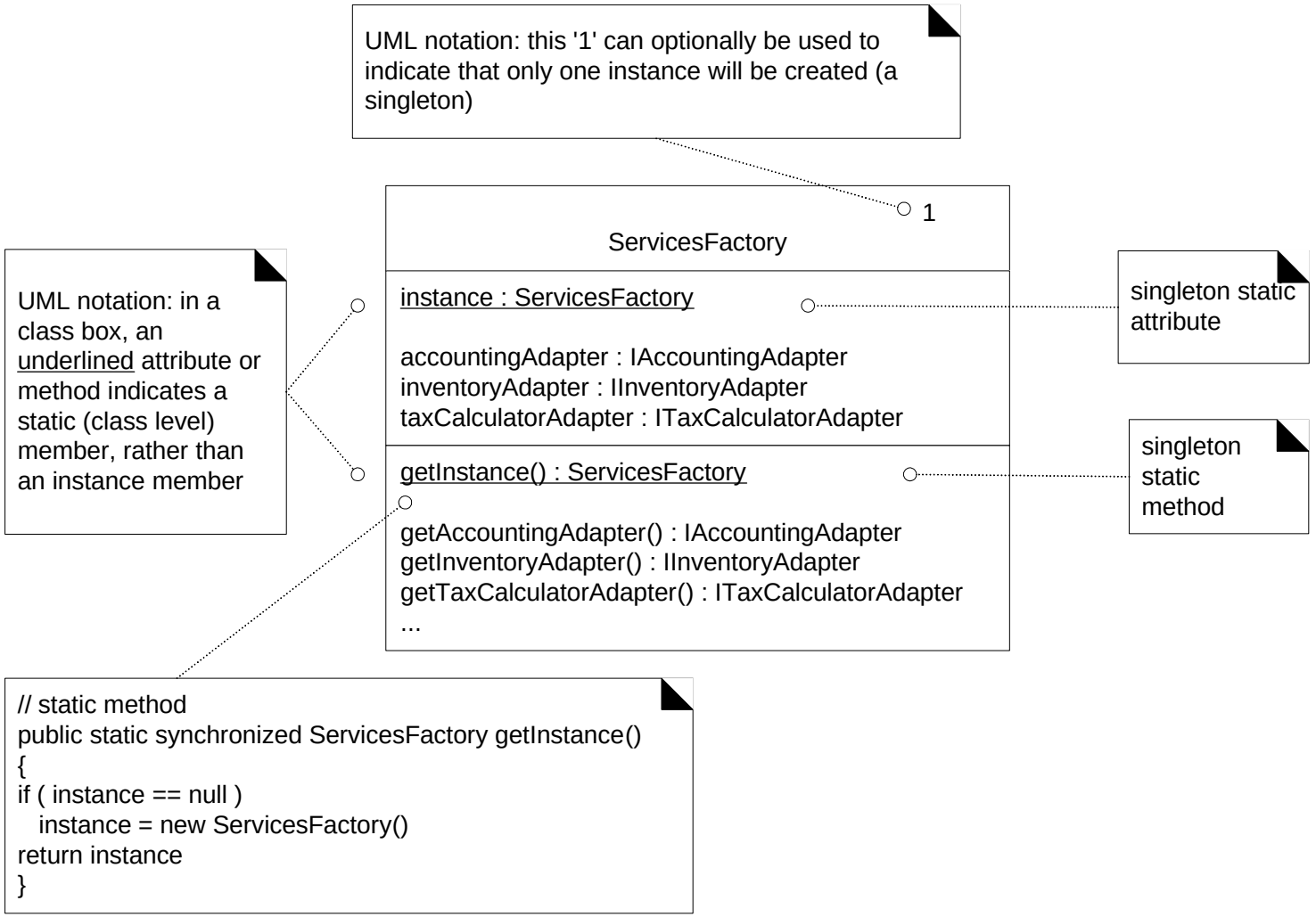
- ¿Quién crea la factoría de servicios y cómo se accede?
  - Sólo se necesita una instancia de la factoría en el proceso.
  - ¿Cómo conseguir visibilidad a la factoría?

$\Rightarrow$ *Es preciso mantener visibilidad global a una única instancia de una clase*

Singleton (GoF)

- Nombre: Singleton
- Contexto / Problema: Se admite exactamente una instancia de una clase –es un “singleton”. Los objetos necesitan un único punto de acceso global.
- Solución (consejo): Definir un método estático de la clase que devuelva el singleton.

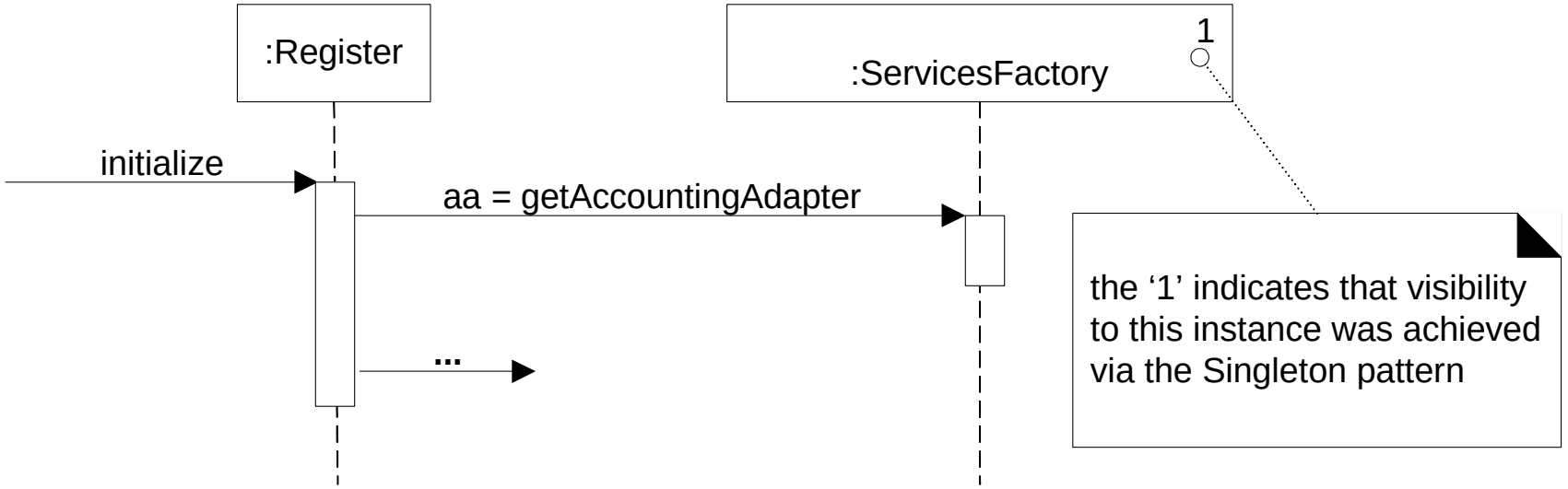
Singleton


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
...
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
singleton static attribute
singleton static method
// static method
public static synchronized ServicesFactory getInstance()
{
if (instance == null)
instance = new ServicesFactory()
return instance
}

Singleton

```
public class Register
{
public void initialize()
{
   ...hace algún trabajo...

   // acceso a la Factoría Singleton mediante la llamada
   // a “getInstancia”

   accountingAdapter =
        ServicesFactory.getInstance().getAccountingAdapter();

   ...hace algún trabajo...
}
// otros métodos
}
```

Singleton

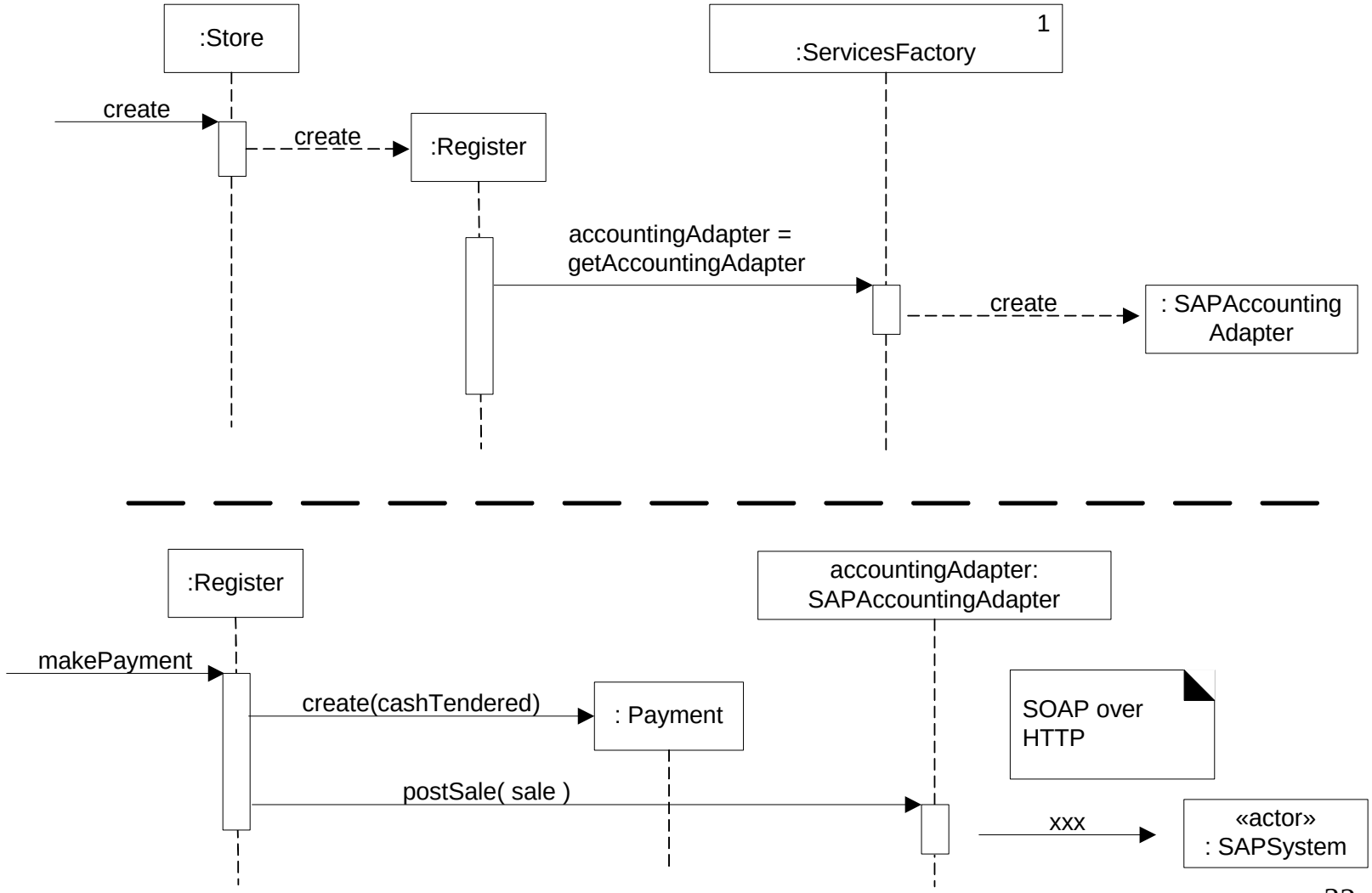

:Register
:ServicesFactory
1
initialize
aa = getAccountingAdapter
...
the ‘1’ indicates that visibility to this instance was achieved via the Singleton pattern

Singleton

- Usualmente, inicialización *perezosa:*

```
public static synchronized ServicesFactory getInstance()
{
    if ( instance == null )
    {
        // sección crítica si es una aplicación
        // con varios hilos
        instance = new ServicesFactory() ;
    }
    return instance ;
}
```

Servicios externos con diversas interfaces – Visión general



Estrategia

En *NuevaEra*…

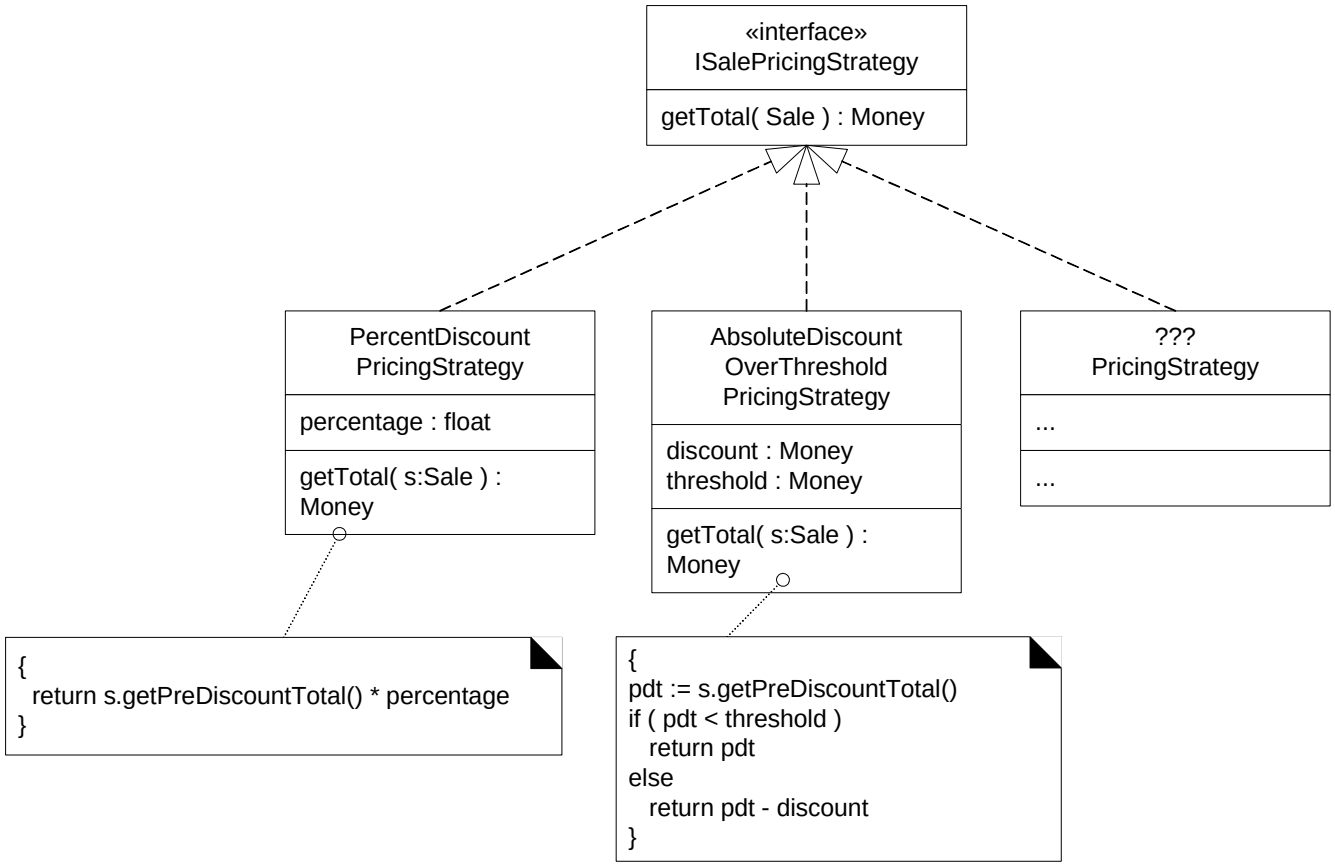
- La estrategia de fijación de precios de una venta puede variar. Durante un periodo podría ser el 10% de descuento en todas las ventas, después podría ser un descuento de 10€ si el total de la venta es superior a 200€, y podrían ocurrir muchas otras variaciones.

  ⇒ *¿Cómo diseñamos los diversos algoritmos de fijación de precios?*

Estrategia (GoF)

- Nombre: Estrategia.
- Contexto/Problema: ¿Cómo diseñar diversos algoritmos o políticas que están relacionadas? ¿Cómo diseñar que estos algoritmos o políticas puedan cambiar?
- Solución (consejo): Definir cada algoritmo/política/estrategia en una clase independiente, con una interfaz común.

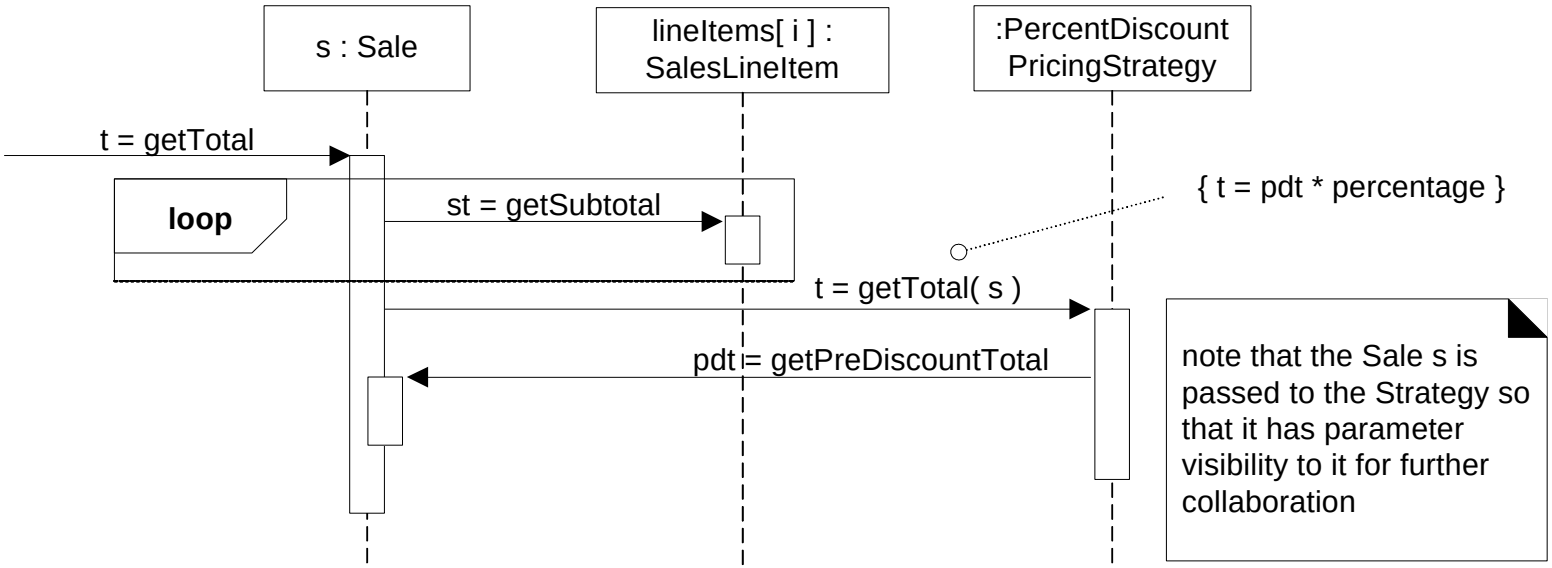
Estrategia


«interface»
ISalePricingStrategy
getTotal(Sale) : Money
PercentDiscount
PricingStrategy
percentage : float
getTotal(s:Sale) :
Money
AbsoluteDiscount
OverThreshold
PricingStrategy
discount : Money
threshold : Money
getTotal(s:Sale) :
Money
???
PricingStrategy
...
...
{
return s.getPreDiscountTotal() * percentage
}
{
pdt := s.getPreDiscountTotal()
if (pdt < threshold)
return pdt
else
return pdt - discount
}

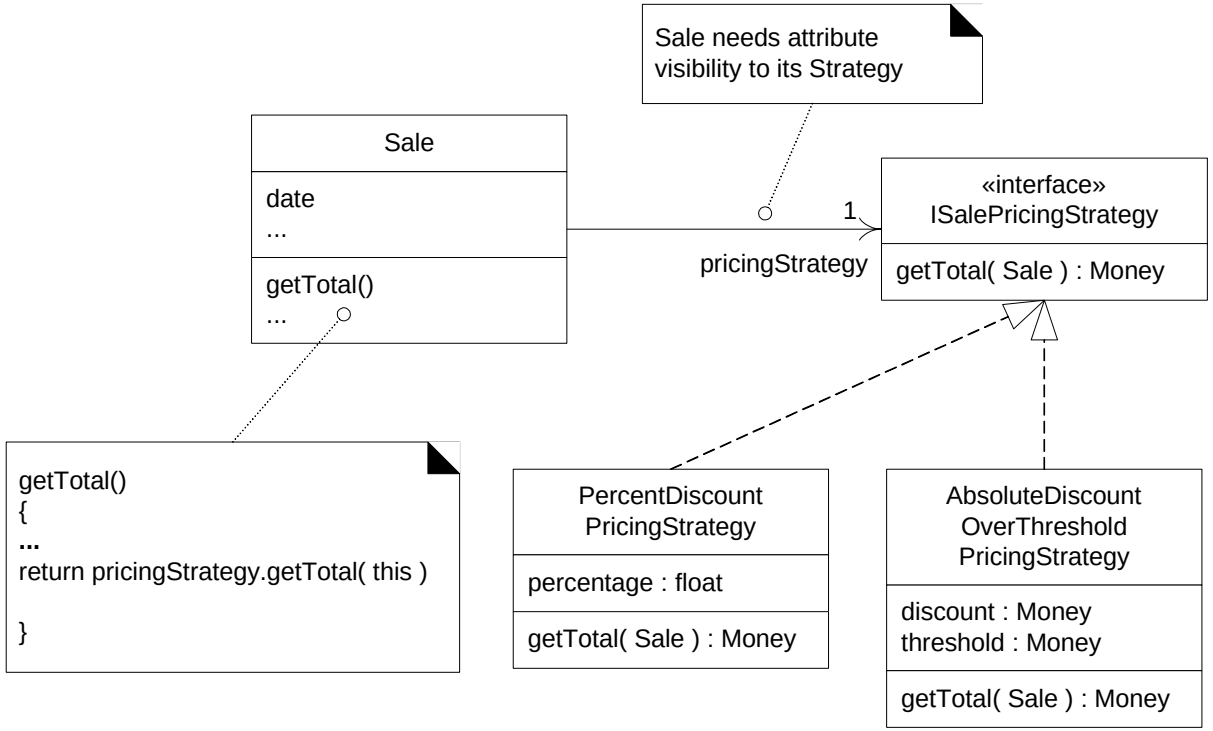
Estrategia

- Un *objeto estrategia* se conecta a un *objeto de contexto* –el objeto al que se aplica el algoritmo, *Venta* en este caso.
- Es habitual que el objeto de contexto pase una referencia a él mismo (*this*) al objeto estrategia.
- El objeto de contexto necesita tener visibilidad de atributo de su estrategia.

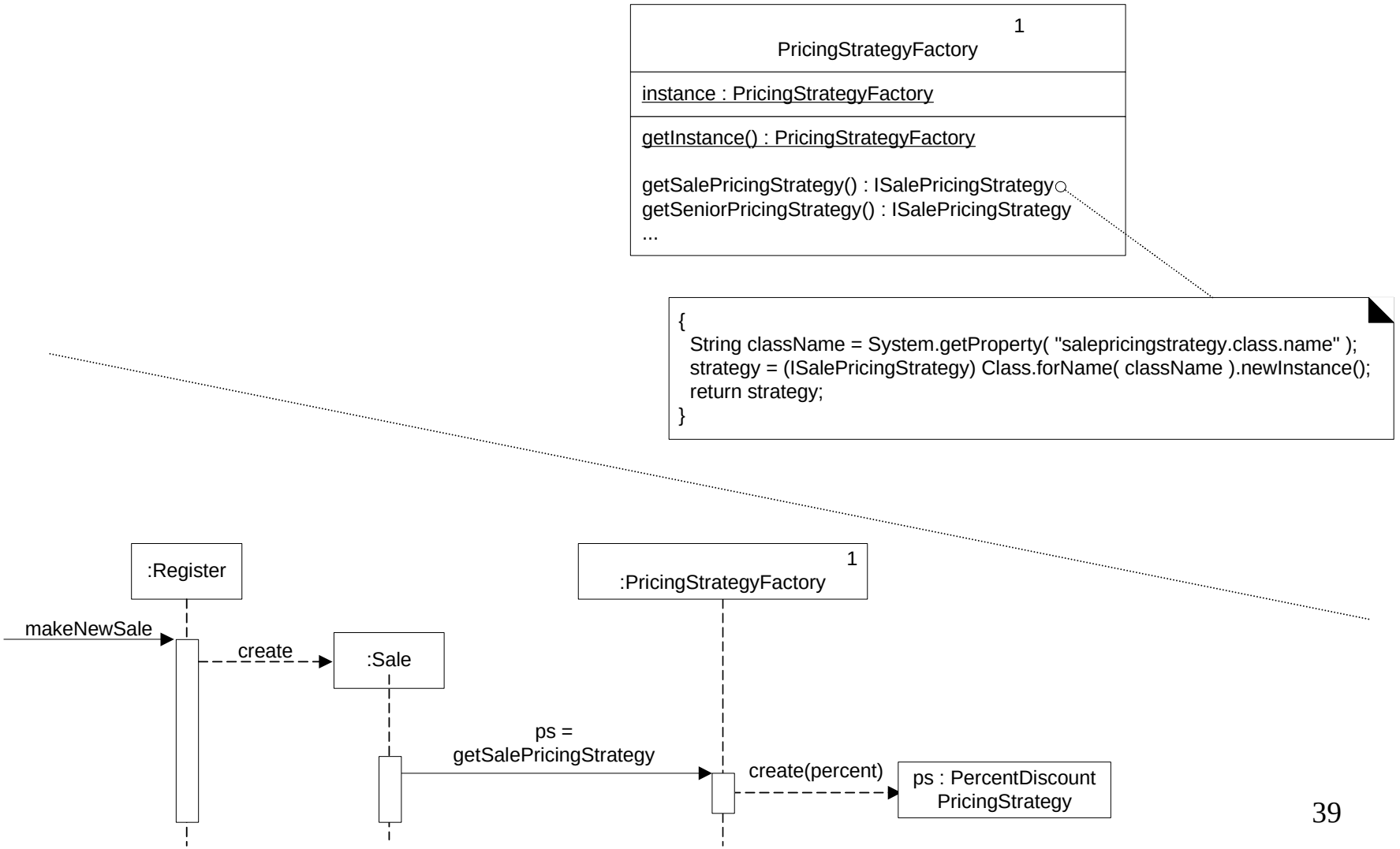
Estrategia


s : Sale
lineItems[i] :
SalesLineItem
:PercentDiscount
PricingStrategy
t = getTotal
loop
st = getSubtotal
{ t = pdt * percentage }
t = getTotal(s)
pdt = getPreDiscountTotal
note that the Sale s is passed to the Strategy so that it has parameter visibility to it for further collaboration

Estrategia


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

Creación de una Estrategia mediante una Factoría



Composite

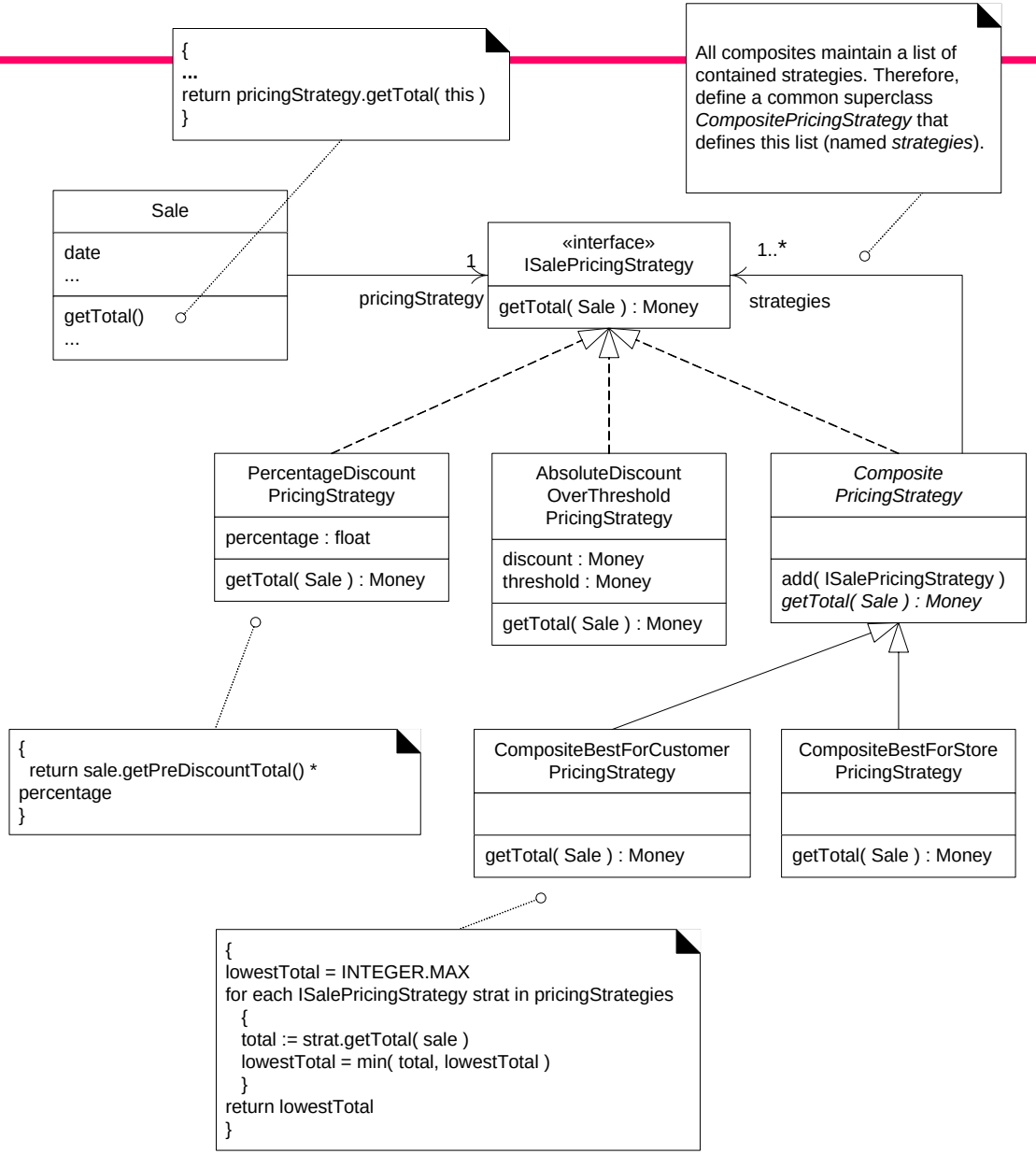
En *NuevaEra*…

- ¿Cómo gestionar el caso de varias políticas contradictorias de fijación de precios? En un momento dado pueden coexistir múltiples estrategias, según el periodo de tiempo, el tipo de producto, o el tipo de cliente.
- Se debe definir la estrategia de resolución de conflictos de la tienda (normalmente, se aplica “lo mejor para el cliente”, pero no es obligatorio).
- Necesitamos cambiar el diseño de forma que el objeto *Venta* no conozca si está tratando con una o más estrategias, y que se resuelvan los conflictos.

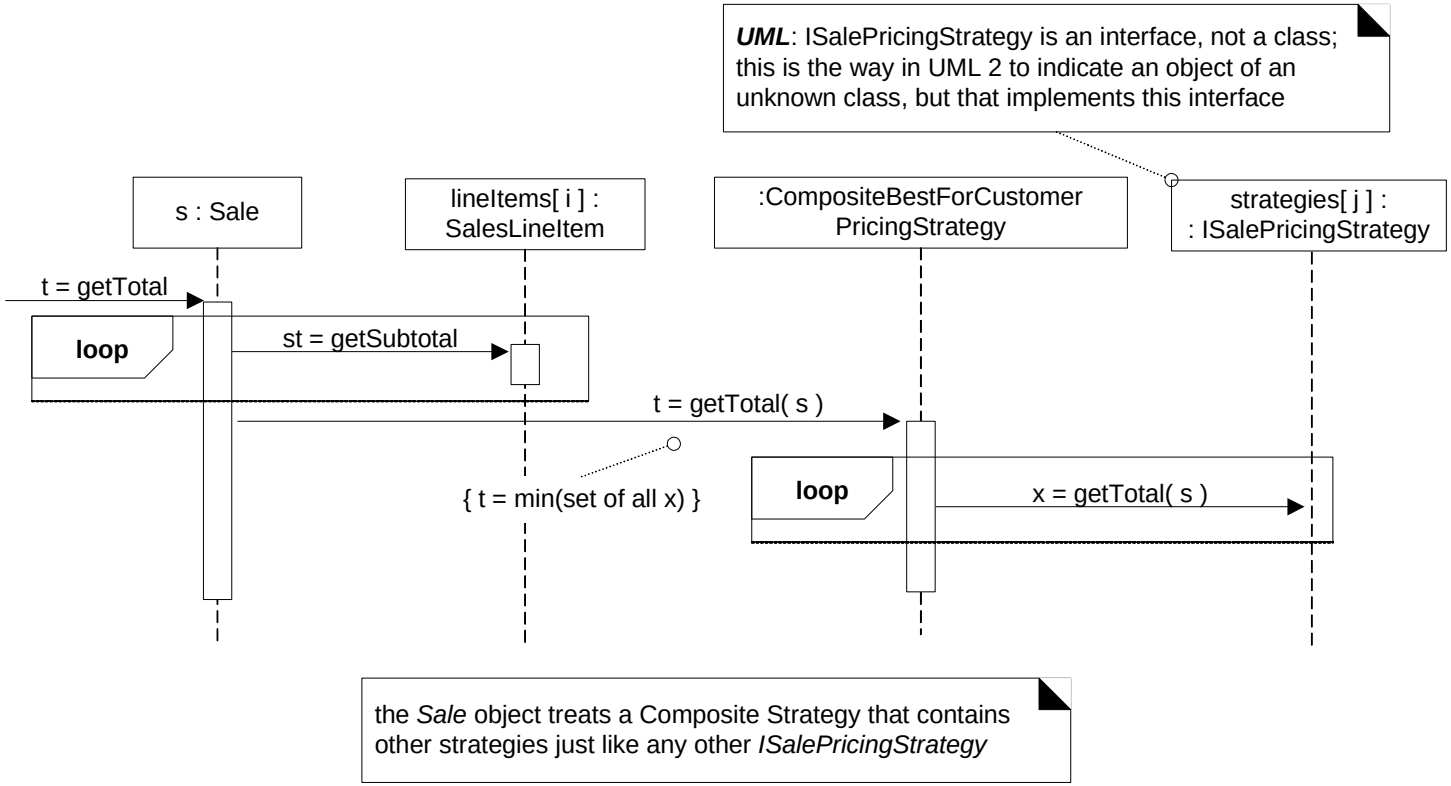
Composite (GoF)

- Nombre: Composite
- Contexto/Problema: ¿Cómo tratar un grupo o una estructura compuesta del mismo modo (*polimórficamente*) que un objeto no compuesto (atómico)?
- Solución (consejo): Definir las clases para los objetos compuestos y atómicos de manera que implementen el mismo interfaz.

Composite

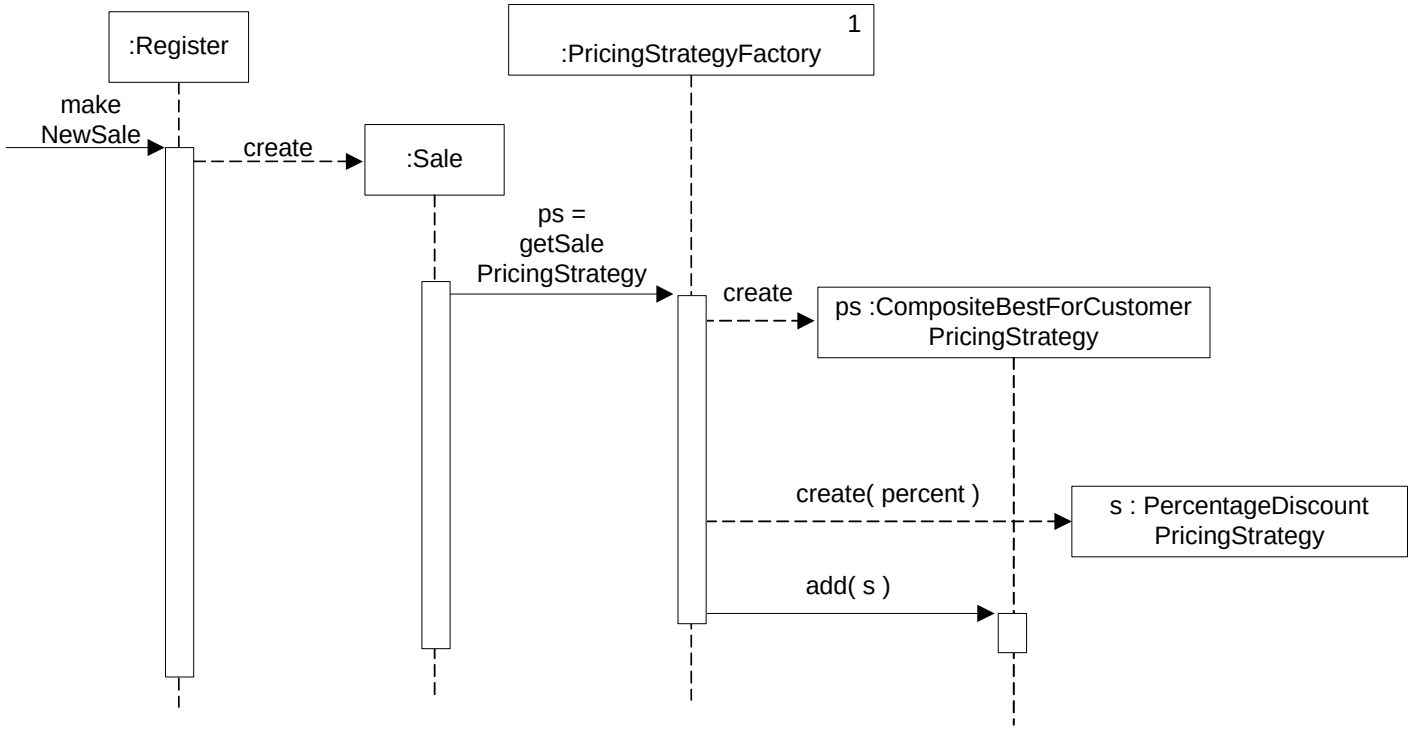

{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).
Sale
date
...
getTotal()
...
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
strategies
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() *
percentage
}
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
}

Composite


UML: ISalePricingStrategy is an interface, not a class; this is the way in UML 2 to indicate an object of an unknown class, but that implements this interface
s : Sale
lineItems[i] : SalesLineItem
:CompositeBestForCustomer PricingStrategy
strategies[j] : : ISalePricingStrategy
t = getTotal
loop
st = getSubtotal
t = getTotal(s)
{ t = min(set of all x) }
loop
x = getTotal(s)
the *Sale* object treats a Composite Strategy that contains other strategies just like any other *ISalePricingStrategy*

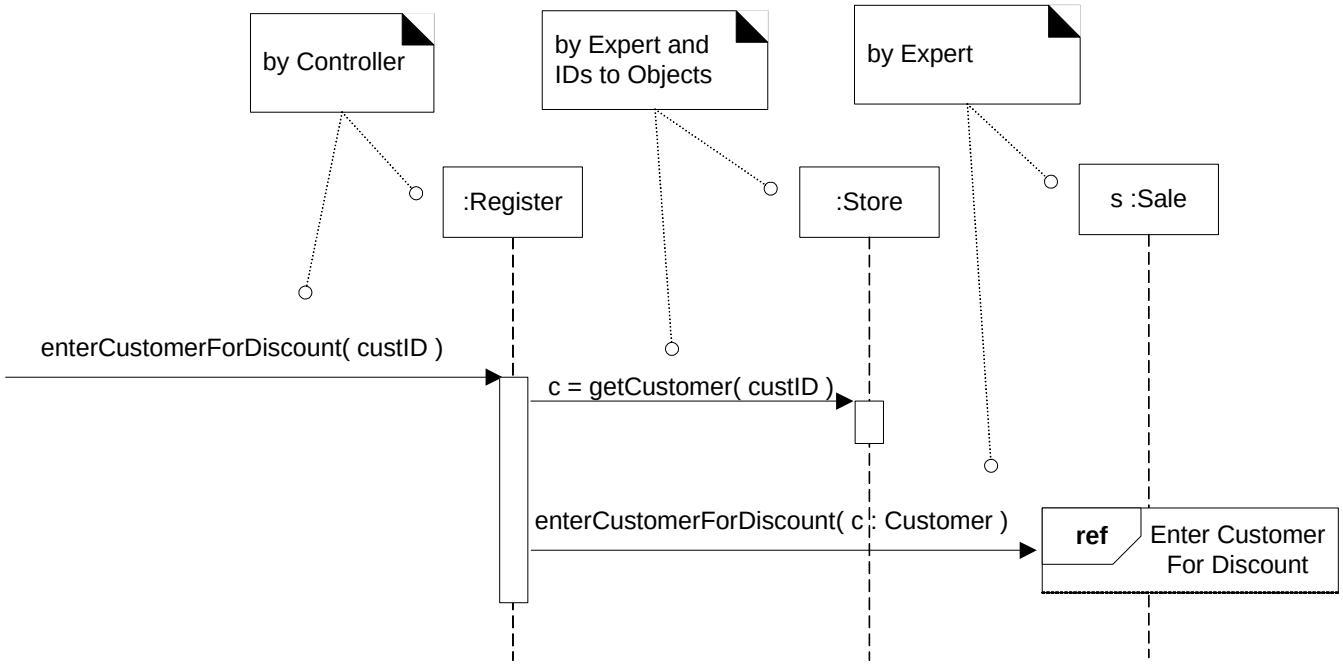
Creación de múltiples instancias
EstrategiaFijarPreciosVenta

- Crear un Composite que contenga las políticas de descuento de la tienda en el momento actual (inicialmente, 0% si no hay ninguna activa).

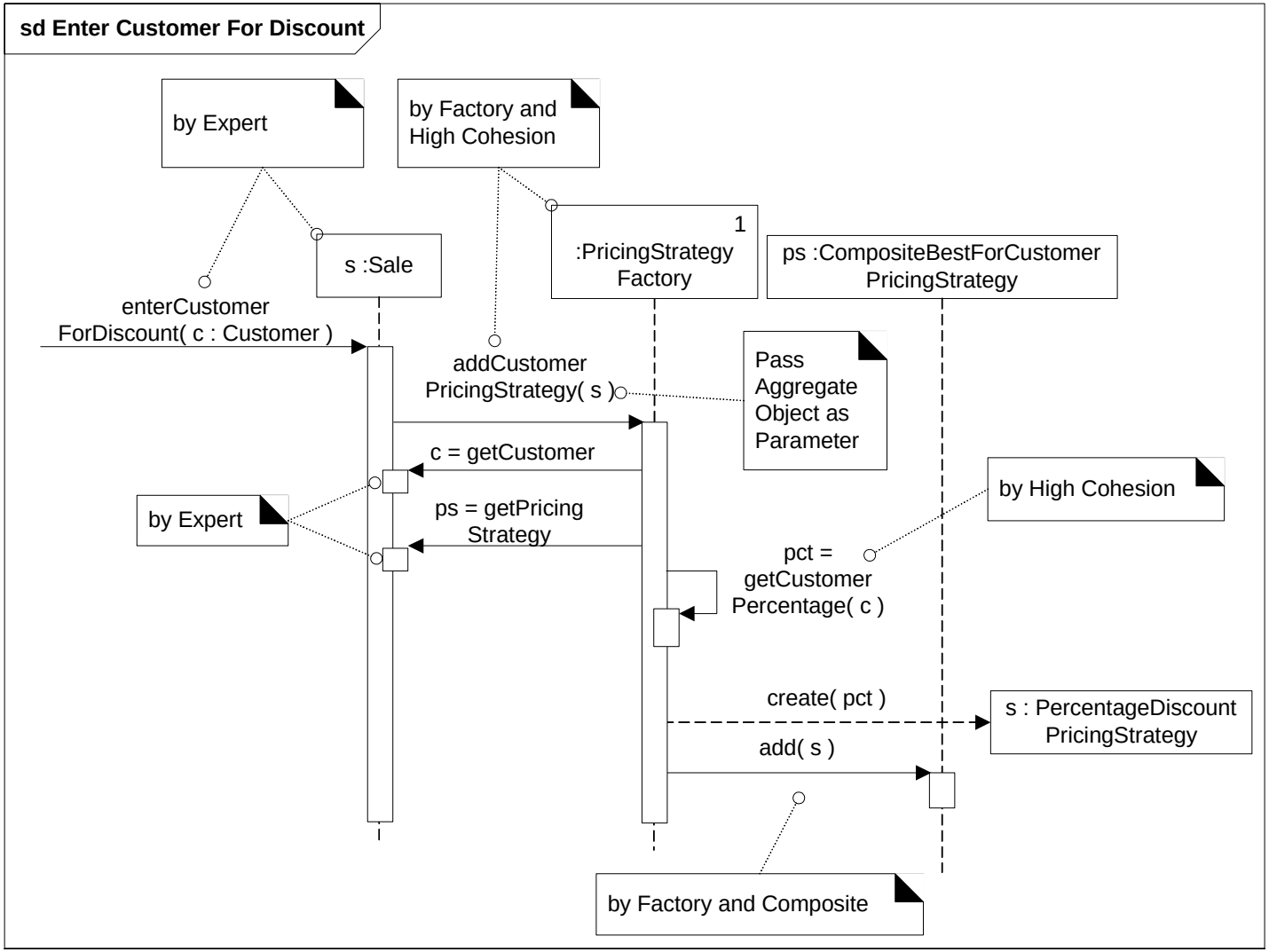


Creación de múltiples instancias
EstrategiaFijarPreciosVenta

- Para introducir el descuento según el tipo de cliente, es preciso introducir un nueva operación del sistema:



Creación de múltiples instancias
EstrategiaFijarPreciosVenta



Fachada

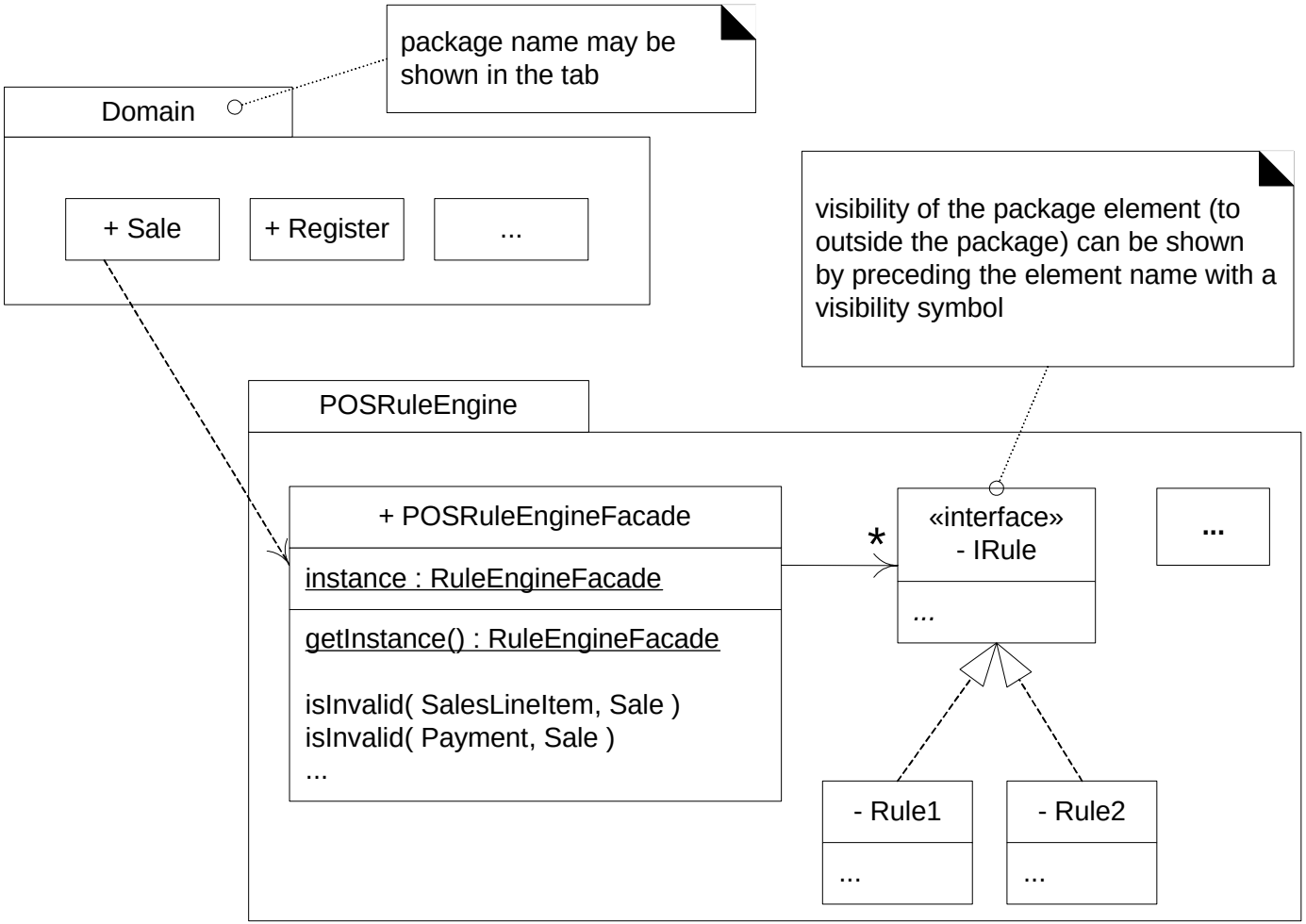
En *NuevaEra*…

- Se quiere dar soporte a *reglas de negocio conectables*.
- Se desea definir un subsistema “motor de reglas” que será responsable de evaluar un conjunto de reglas contra una operación, e indicar si alguna de las reglas invalida la operación.

Fachada (GoF)

- Nombre: Fachada (*Facade*)
- Contexto/Problema: Se requiere una interfaz común, unificada para un conjunto de implementaciones o interfaces dispares –como en un subsistema. Podría no ser conveniente acoplarse con muchas cosas del subsistema, o la implementación del subsistema podría cambiar. ¿Qué hacemos?
- Solución (consejo): Definir un punto único de conexión con el subsistema –un objeto fachada que envuelve al subsistema. Este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los componentes del subsistema.

Fachada


package name may be shown in the tab
Domain
+ Sale
+ Register
...
visibility of the package element (to outside the package) can be shown by preceding the element name with a visibility symbol
POSRuleEngine
+ POSRuleEngineFacade
instance : RuleEngineFacade
getInstance() : RuleEngineFacade
isInvalid(SalesLineItem, Sale)
isInvalid(Payment, Sale)
...
*
«interface»
- IRule
...
...
- Rule1
...
- Rule2
...

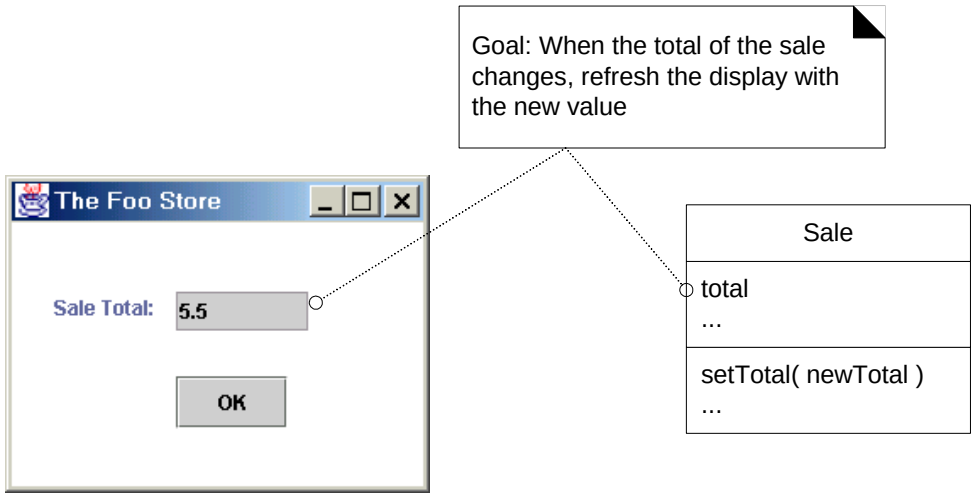
Fachada

```
public class Venta
{

public void crearLineaDeVenta( EspecificacionDelProducto espec, int
   cantidad)
{
   LineaDeVenta ldv = new LineaDeVenta( espec, cantidad );

   // llamada a la fachada, obsérvese el uso de Singleton
   if (FachadaMotorReglasPDV.getInstancia().esInvalido( ldv, this ))
         return;

   lineasDeVenta.add( ldv );
}
//…
} // final de la clase
```

Observador

En *NuevaEra*…

- Se pretende añadir la capacidad de que una ventana GUI actualice la información que muestra sobre el total de la venta cuando éste cambia.



Observador (GoF)

- Nombre: Observador / Observer / Publicar-Suscribir / Modelo de delegación de eventos
- Contexto/Problema: Diferentes tipos de objetos suscriptores están interesados en el cambio de estado o eventos de un objeto emisor, y quieren reaccionar cada uno a su manera cuando el emisor genere un evento. Además, el emisor quiere mantener bajo acoplamiento con los suscriptores. ¿Qué hacemos?
- Solución (consejo): Definir un interfaz “suscriptor” u “oyente” (*listener*). Los suscriptores implementan este interfaz. El emisor dinámicamente puede registrar suscriptores que están interesados en un evento, y notificarles cuando ocurre un evento.

Observador

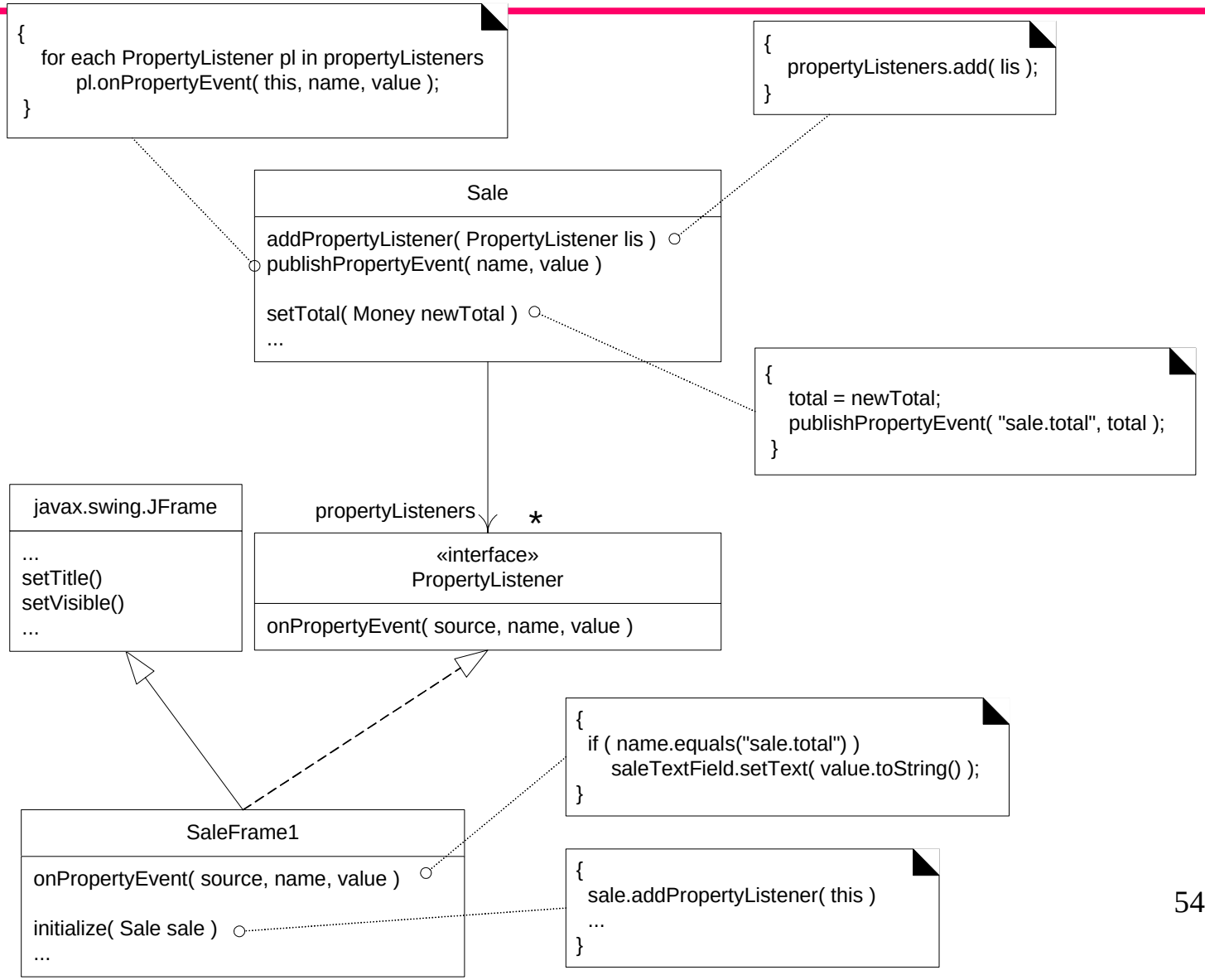
En *NuevaEra…¿porqué no vale la siguiente solución?*

"Cuando la Venta cambia su total, el objeto Venta envía un mensaje a la ventana, pidiéndole que actualice la información que muestra."

⇒El principio de separación Modelo-Vista disuade de tales soluciones. Los objetos del modelo no deberían conocer los objetos de la vista o presentación.

⇒De esa manera, se permite reemplazar la vista o capa de presentación por una nueva sin afectar a los objetos que no pertenecen a la interfaz de usuario.

Observador


{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

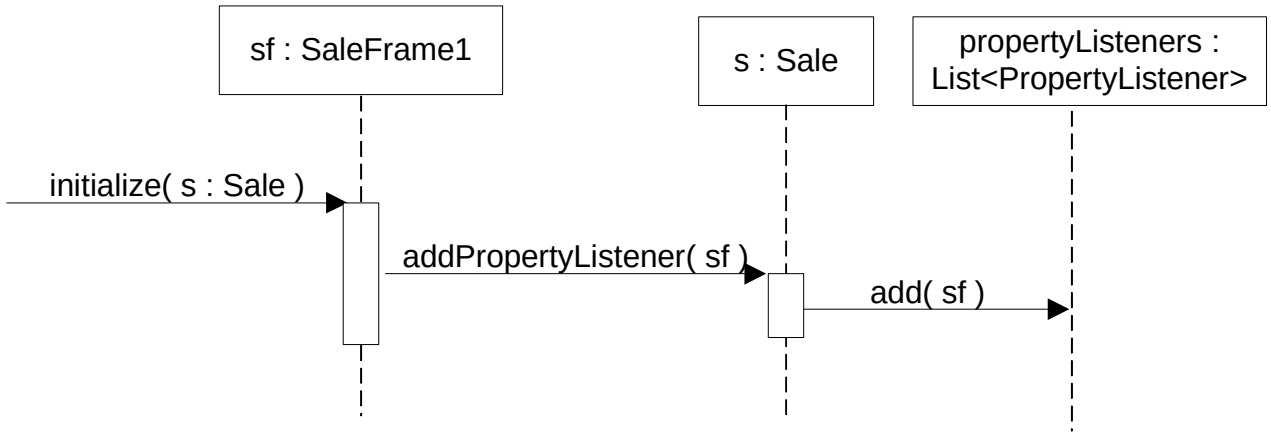
Observador

Ideas y pasos fundamentales:

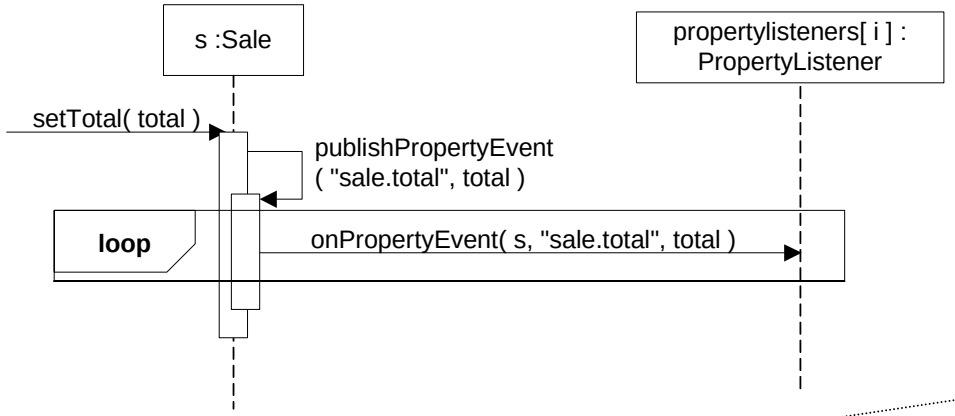
1. Se define una interfaz; en este caso, *PropertyListener* con la operación *onPropertyEvent.*
2. Se define la ventana que implementa la interfaz
   SaleFrame1 implementará el método *onPropertyEvent.*
3. Cuando se inicializa la ventana *SaleFrame1*, se le pasa la instancia de *Sale* de la cual está mostrando el total.
4. La ventana *SaleFrame1* se registra o suscribe a la instancia de Sale para que le notifique acerca de los eventos sobre la propiedad, por medio del mensaje *addPropertyListener*. Es decir, cuando una propiedad (como *total*) cambia, la ventana quiere que se le notifique.
5. Obsérvese que *Sale* no conoce los objetos *SaleFrame1*; más bien sólo conoce los objetos que implementan la interfaz *PropertyListener*. Esto disminuye el acoplamiento entre la Venta y la ventana –se acopla sólo con una interfaz, no con la clase de la GUI.
6. Por tanto, la instancia de *Sale* es un *emisor* de “eventos sobre la propiedad”. Cuando cambia el total, itera sobre todos los objetos *PropertyListener* que están suscritos, y se lo notifica a cada uno.

Observador

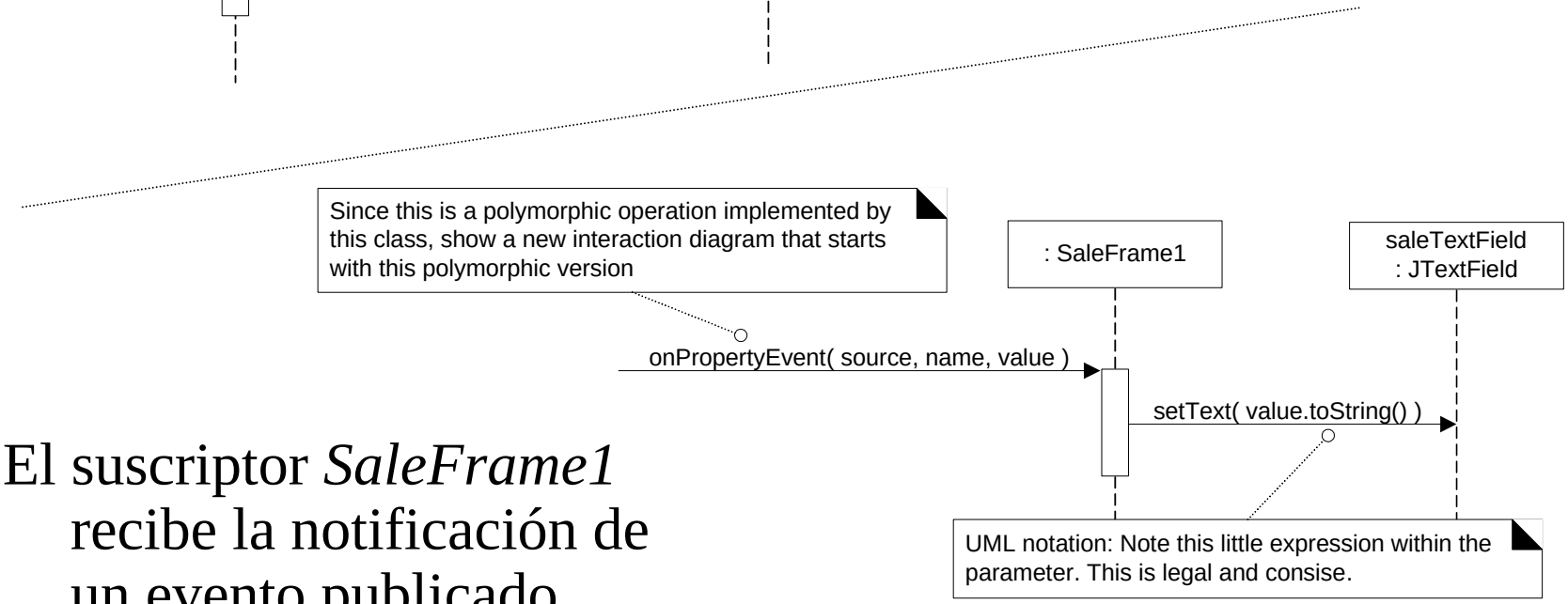
El observador *SaleFrame1* se suscribe al emisor *Sale*:



Observador



Sale publica un evento sobre la propiedad a todos sus suscriptores



El suscriptor *SaleFrame1* recibe la notificación de un evento publicado