Análisis y Diseño de Sistemas

Tema: Proceso RUP Agil
Diseño de Software para el reuso III
Aplicación de Patrones al Ejemplo TPDV

Mg. Ing. Adriana Urciuolo

UNTDF, 2015

Agenda

- Uso de patrones en la Fase de Elaboración.
- Requerimientos y Dominio en la Iteración2
  - Repaso de Conceptos
  - Ejemplo
- Aplicación de Patrones GoF:
  - Adapter
  - Factory
  - Singleton
  - Strategy
  - Composite

Fase de Elaboración: La Segunda Iteración

- Durante la Fase de Inicio y la primera iteración de la Fase de Elaboración, el énfasis se puso en ls actividades fundamentales de análisis y diseño.
- En la Segunda Iteración, el énfasis se pondrá en el refinamiento del Modelo de Diseño:
  - El uso de Patrones de diseño para crear una solución de diseño sólida

Aplicando Patrones durante el Diseño:

- Patrones GRASP
- Patrones Gof

Fase de Elaboración

Requerimientos y Dominio en la Iteración 2

Requisitos en la Iteración 2 del ejemplo PDV

- La iteración 2 del Ejemplo PDV maneja varios requisitos:
  - Soporte a las variaciones de servicios externos de terceras partes. Se deben conectar al sistema diferentes calculadores de impuestos, cada uno con su propia interfaz. Igualmente con diferentes sistemas de contabilidad, etc. Cada uno ofrecerá un API y un protocolo diferentes.
  - Reglas de fijación de precios complejas
  - Reglas de negocio conectables
  - Un diseño para actualizar una ventana del GUI cuando cambia el total de la venta

  Solo se tendrán en cuenta estos requisitos (en esta iteración) en el contexto de los escenarios del CU *Procesar Venta*

Requisitos en la Iteración 2 del ejemplo PDV

- Los requisitos mencionados no son nuevos, sino detectados durante la Fase de Inicio
- Por ej., el CU Procesar Venta señala la cuestión de fijación de precios:
- **Escenario Principal de Exito**
  1. El cliente llega a un terminal PDV con mercancías que comprar.
  2. El Cajero le dice al Sistema que cree una nueva venta
  3. El Cajero introduce el identificador de Artículo
  4. El Sistema registra la línea de la venta y presenta la descripción del artículo, precio y suma parcial. ***El precio se calcula a partir de un conjunto de reglas de precio.***
  5. ***……***

Requerimientos no funcionales

- En la Especificación suplementaria (req. no funcionales):
  - ✓ Se registran los detalles de reglas de dominio para los precios y
  - ✓ Se señala la necesidad de soportar varios sistemas externos

...

Interfaces

Interfaces software

Para la mayoría de los sistema externos de colaboración (calculador de impuestos, contabilidad, inventario,...) necesitamos ser capaces de conectar diversos sistemas y, por tanto, diversas interfaces.

...

Reglas de Dominio (Negocio)

ID	*Regla*	*Grado de variación*	*Fuente*
REGLA4	Reglas de descuento al comprador. Ejemplos: Empleado—20% de descuento Clientes preferentes—10% de descuento Antiguos—15% de descuento	Alto. Cada tienda utiliza reglas diferentes.	Política de la tienda.
...	...	...	...

Información en dominios de interés

Fijación de precios

Además de las reglas de fijación de precios que se describen en la sección de reglas del dominio, nótese que los artículos tienen un *precio original*, y opcionalmente, un *precio rebajado*. El precio de los artículos (antes de los descuentos adicionales) es el precio rebajado, si existe. Las organizaciones mantienen el precio original incluso si hay un precio rebajado, por razones de contabilidad e impuestos.

Desarrollo incremental del CU Procesar Venta

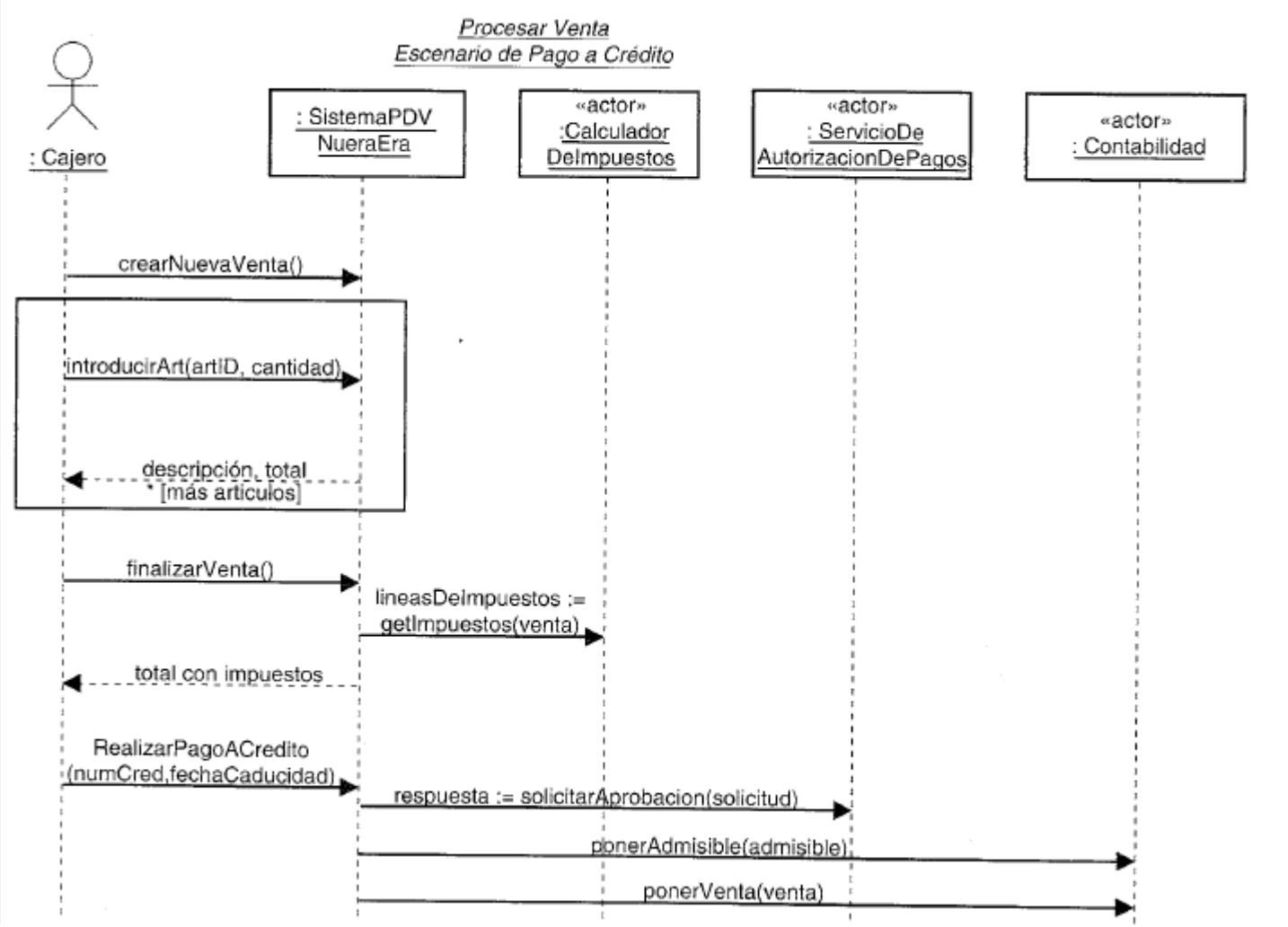
- En la 2a iteración estamos implementando más escenarios del CU *Procesar Venta* de manera que el sistema crece incrementalmente
  - ✓ Se trabaja con varios escenarios o características del mismo CU a lo largo de varias iteraciones
  - ✓ Se extiende el sistema gradualmente para finalmente manejar toda la funcionalidad requerida
  - ✓ Los CU breves y simples pueden implementarse en una iteración
- En la Iteración 1 se hicieron simplificaciones para abordar la solución sin complejidad
- Por ello se considera pequeña cantidad de funcionalidad adicional

Refinamiento de artefactos de Análisis

- Modelo de Casos de uso. Casos de Uso:
  - ✓ No se requiere refinamiento de los CU en esta iteración.
- Modelo de Casos de Uso. DSS
  - ✓ Esta iteración abarca la inclusión del soporte para sistemas externos de terceras partes con varias interfaces, como un calculador de impuestos.
  - ✓ El Sistema PDV se comunicará de manera remota con los sistemas externos.
  - ✓ Deben actualizarse los DSS para reflejar las colaboraciones inter-sistemas, mostrando los nuevos eventos.
- Modelo de Dominio
  - ✓ Los requisitos que se abordan en esta iteración no comprenden muchos conceptos del dominio nuevos
  - ✓ Un concepto nuevo puede ser *ReglaDePrecio*
- Modelo de Casos de Uso. Contratos: No se consideran nuevos.

Refinamiento de artefactos de Análisis: DSSs

- DSS: Escenario de Pago a Crédito que ilustra eventos externos:



Fase de Elaboración
2ª Iteración

Diseño de Objetos para la realización de Casos de Uso aplicando Patrones

Diseño p/Requisito

- **Caso de uso Procesar Venta**:
- Soporte para el acceso a los servicios externos de terceras partes, cuyas interfaces podrían variar

Patrón Adapter

- **Problema**: Cómo resolver interfaces incompatibles o proporcionar una interfaz estable para componentes parecidos con distintas interfaces?

- **Solución**: Convierta la interfaz original de un componente en otra interfaz mediante un objeto adaptador intermedio.

Patrón Adapter y Ejemplo PDV

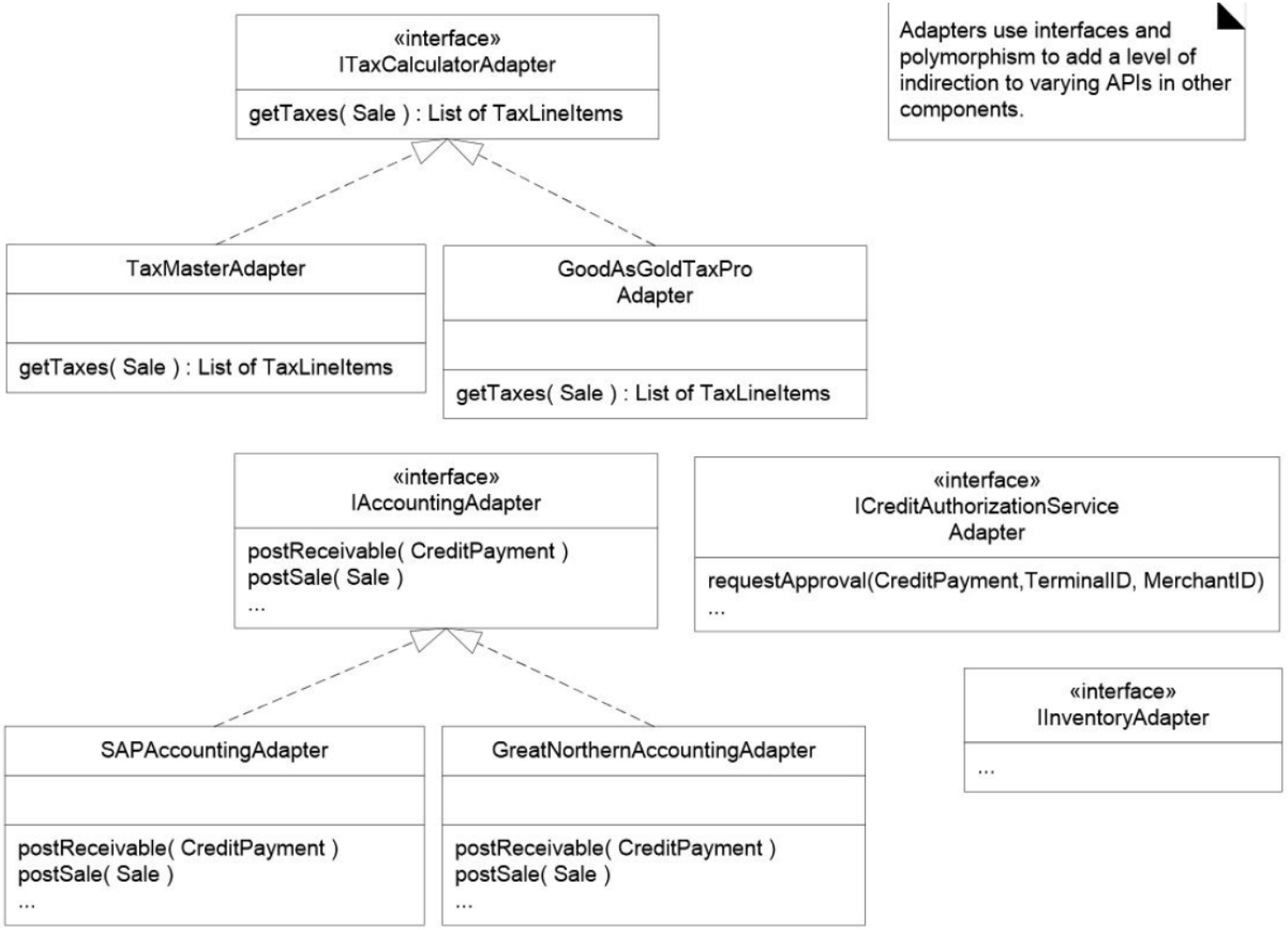
- **Motivación:**

✓ El Sistema de PDV 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, de contabilidad, etc.

✓ Cada uno tiene un API diferente que no puede ser cambiada.

- **Aplicación:** Se añade 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

Patrón Adapter y Ejemplo PDV



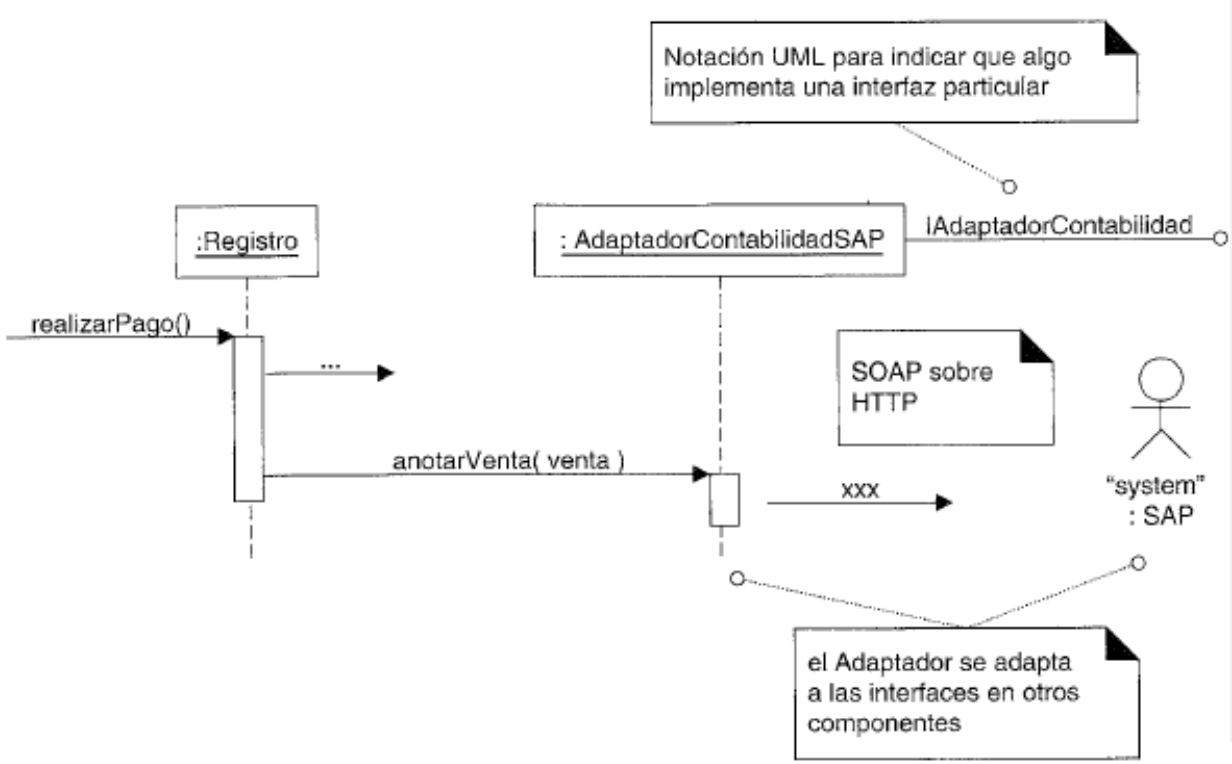
Patrón Adapter y Ejemplo PDV

- **Guía**: Incluir el Patrón en el nombre del Tipo
- Es común 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.

- **Patrones relacionados**
- Un adaptador de recursos que oculta un sistema externo, puede ser considerado también un objeto Fachada (otro patrón GoF), dado que adapta el acceso a un subsistema o sistema dentro de un objeto simple.
- La motivación de llamarlo recurso adaptador se mantiene cuando se provee adaptación a interfaces externas variantes

Patrón Adapter y Ejemplo PDV

- **Ejemplo:** Una instancia particular de Adapter será instanciada para el servicio3 externo elegido, tal como SAP para contabilidad, y adaptará el request *anotarVenta* a la interface externa, tal como la interface SOAP XML sobre HTTPS para un servicio Web de Intranet ofrecido por SAP



GRASP como generalización de Patrones

- El uso previo del patrón Adapter puede verse como una especialización de componentes básicos de GRASP:
  - ✓ Soporta *Variaciones Protegidas* con respecto a cambios en las interfaces externas o paquetes de terceras partes mediante el uso de un Objeto *Indirección* que aplica interfaces y *Polimorfismo*.
- Principios GRASP como solución a la abundancia de patrones para recordar:
  - ✓ Muchos Patrones de diseño pueden verse como especialización de los principios

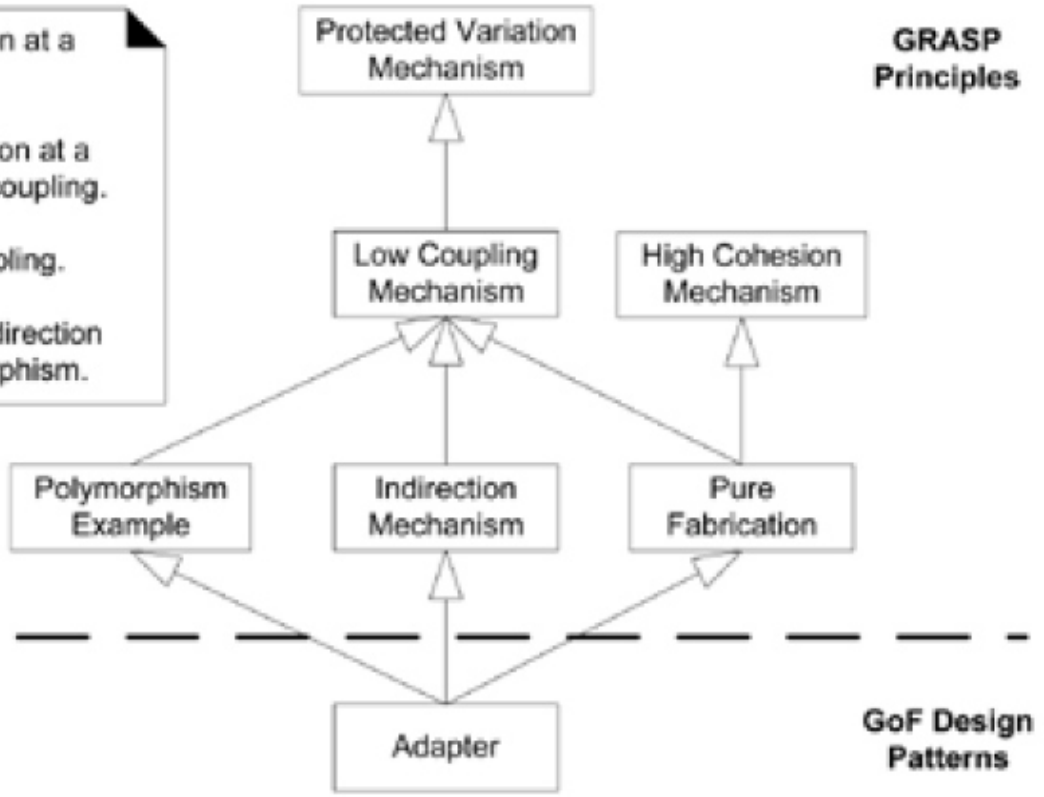
GRASP como generalización de Patrones

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

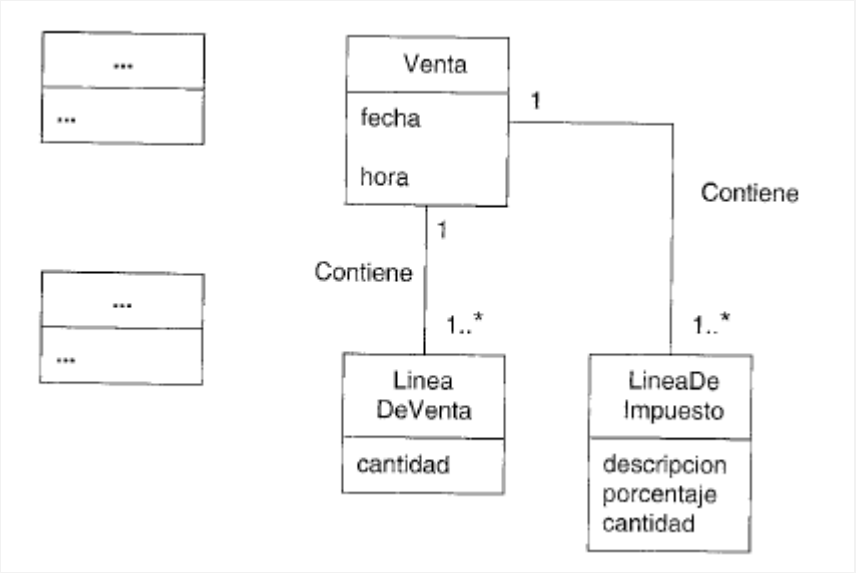
An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



Actualización del Modelo de Dominio

- En el diseño del Adaptador se observa que la operación getImpuestos devuelve una *LineaDeImpuestos,* ya que a una *Venta* normalmente se asocian distintos impuestos
- Además de una clase de diseño es un concepto de Dominio, lo cual lleva a modificar el Modelo de Dominio.
- Este modelo se modifica hasta que tiene sentido hacerlo



Patrón Factory

- El adaptador plantea un nuevo problema en el diseño: En la solución anterior del patrón adaptador para los servicios externos con interfaces diferentes,
  - ✓ ¿Quién crea los adaptadores? y
  - ✓ ¿Cómo determinar qué clase de adaptador crear, como TaxMaster-Adapter o GoodAsGoldTaxProAdapter?

- Si los creara algún objeto del dominio, éstos excederían la lógica pura de la aplicación, para entrar en cuestiones relacionadas con la conexión de componentes SW externos

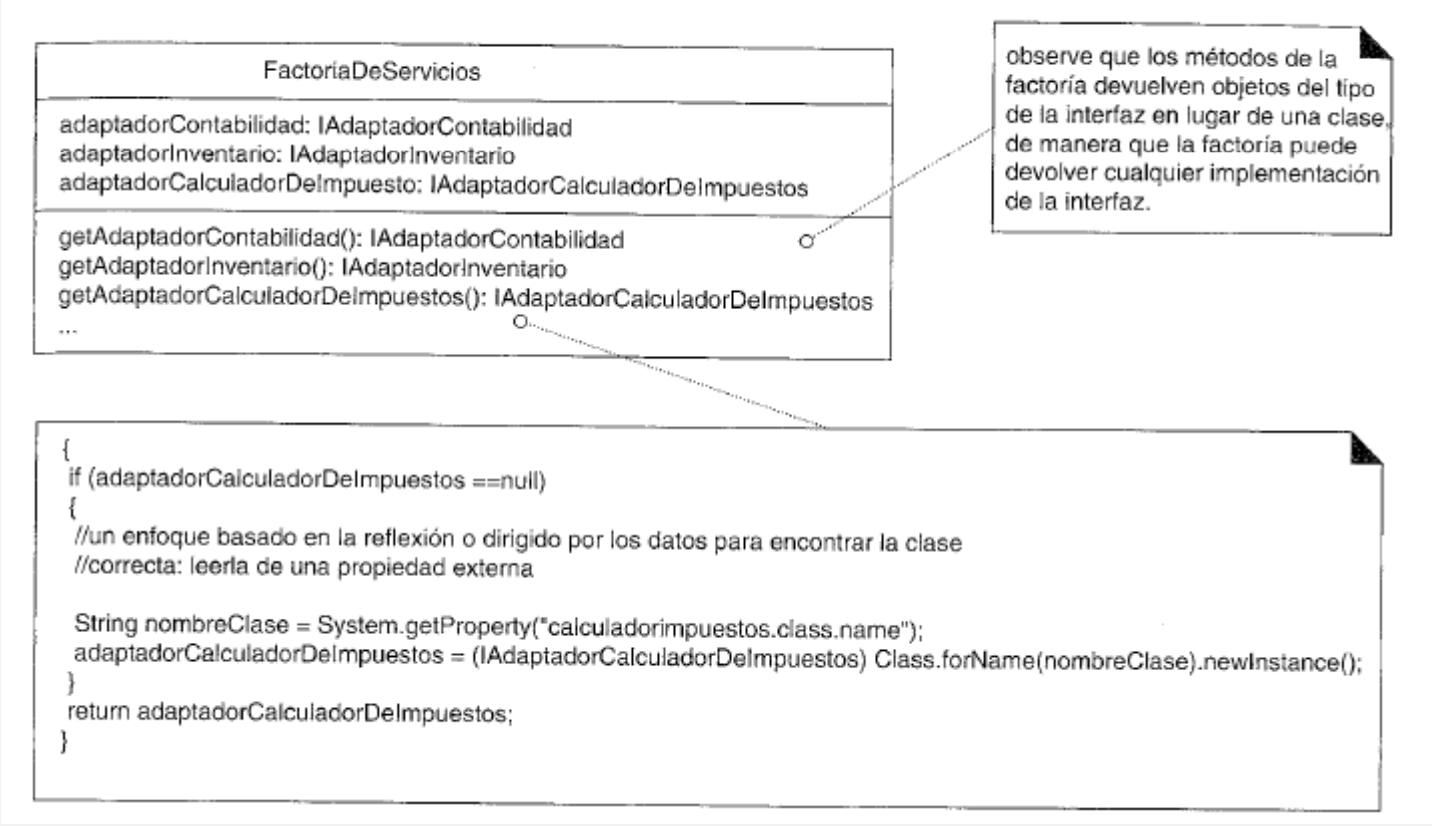
Patrón Factory

- **Problema**: Quién debe ser el responsable de la creación de objetos cuando existen condiciones especiales, como lógica de creación compleja?
- **Solución**: Crear un Objeto Fabricación Pura denominado Factory para manejar la creación de objetos.

- Los objetos de fábrica tienen varias ventajas:
  - Separar la responsabilidad de creaciones complejas en objetos de ayuda (helpers) cohesivos.
  - Ocultar lógica de creación potencialmente compleja.

Patrón Factory

- La lógica para decidir qué clase se crea se resuelve leyendo el nombre de la clase de una fuente externa (por ej. por medio de una propiedad del sistema, si se usa Java), y después se carga la clase dinámicamente



Patrón Singleton

- Con la FactoríaDeServicios del sistema PDV surge:
- Problema de diseño: Quién crea la factoría de servicios y cómo se accede?
  - ✓ Sólo se necesita una instancia de la factoría en el proceso
  - ✓ Se podría necesitar invocar los métodos de la factoría desde varios sitios del código, puesto que en diferentes lugares se necesita acceder a los adaptadores para solicitar los servicios externos
  - ✓ Problema de visibilidad: Cómo conseguir visibilidad a esta una única instancia de FactoriaDeServicios?.
- Alternativa: Patrón *Singleton*

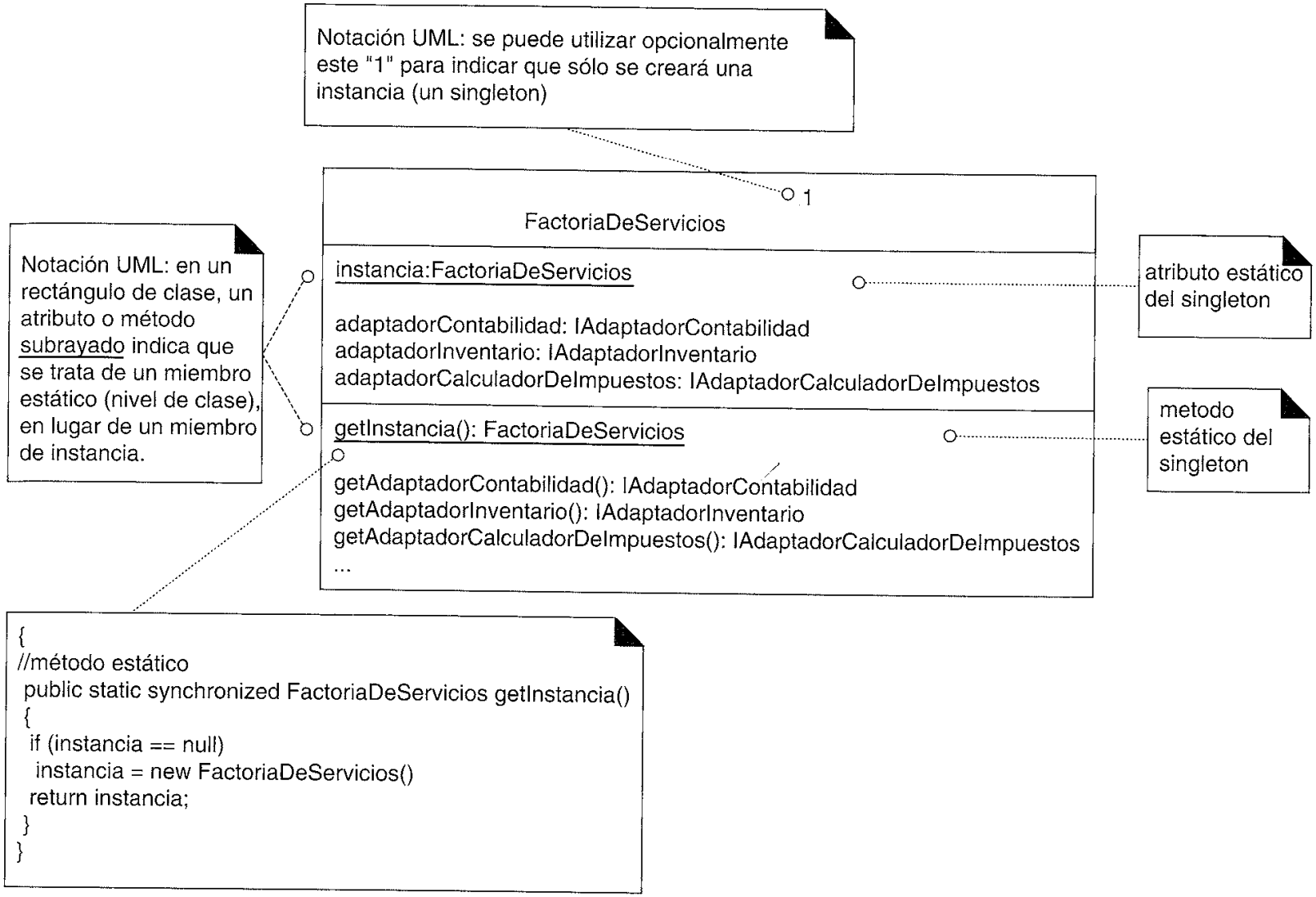
Patrón Singleton

- **Problema:** Se admite exactamente una instancia de una clase, un “singleton”. Los objetos necesitan un único punto de acceso global.

- **Solución**: Definir un método estático de la clase que devuelva el singleton

- La idea clave es que la clase X defina un método estático *getInstancia* que proporciona una instancia única de X.
- Un desarrollador tiene visibilidad global a esta instancia única por medio del método estático de la clase *getInstancia*

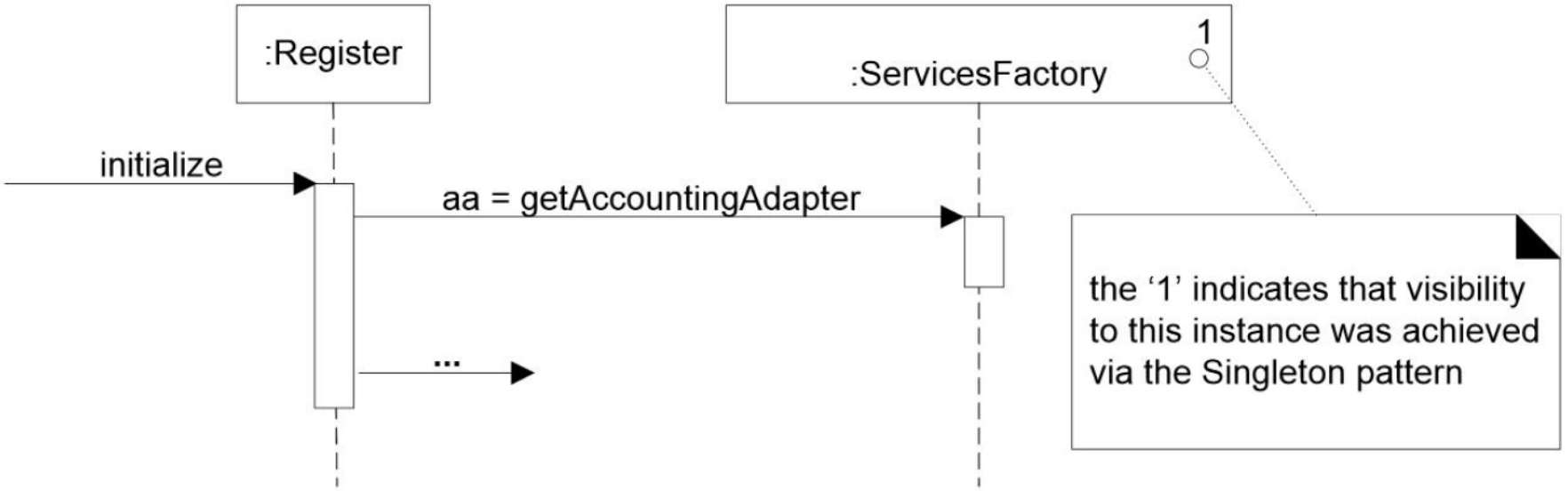
Patrón Singleton



Patrón Singleton

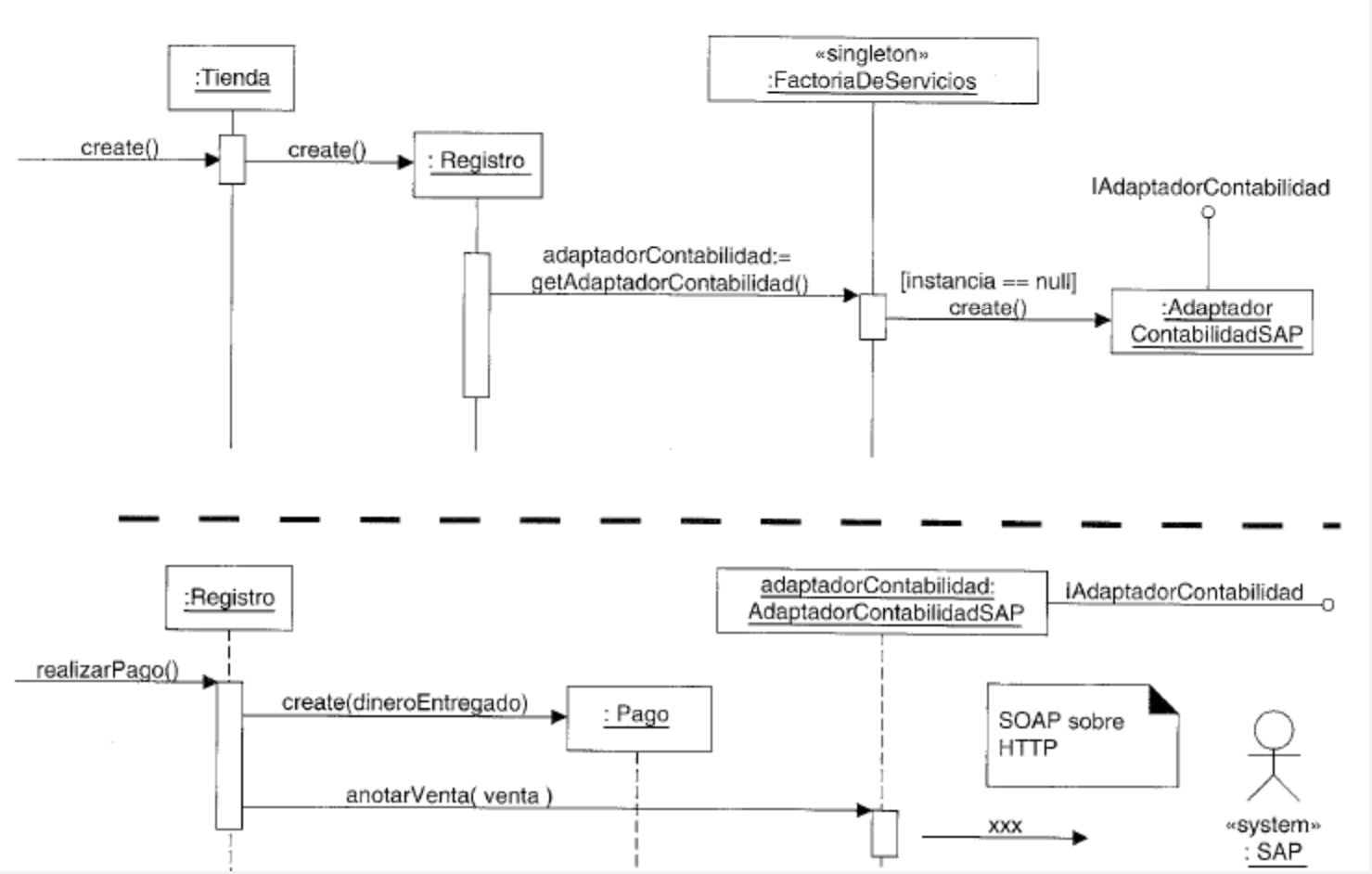
```
public class Register
{
public void initialize()
{
    ...hace algún trabajo...

    // acceso a la Factoría Singleton mediante la llamada
    // a "getInstancia"

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

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

Patrón Singleton

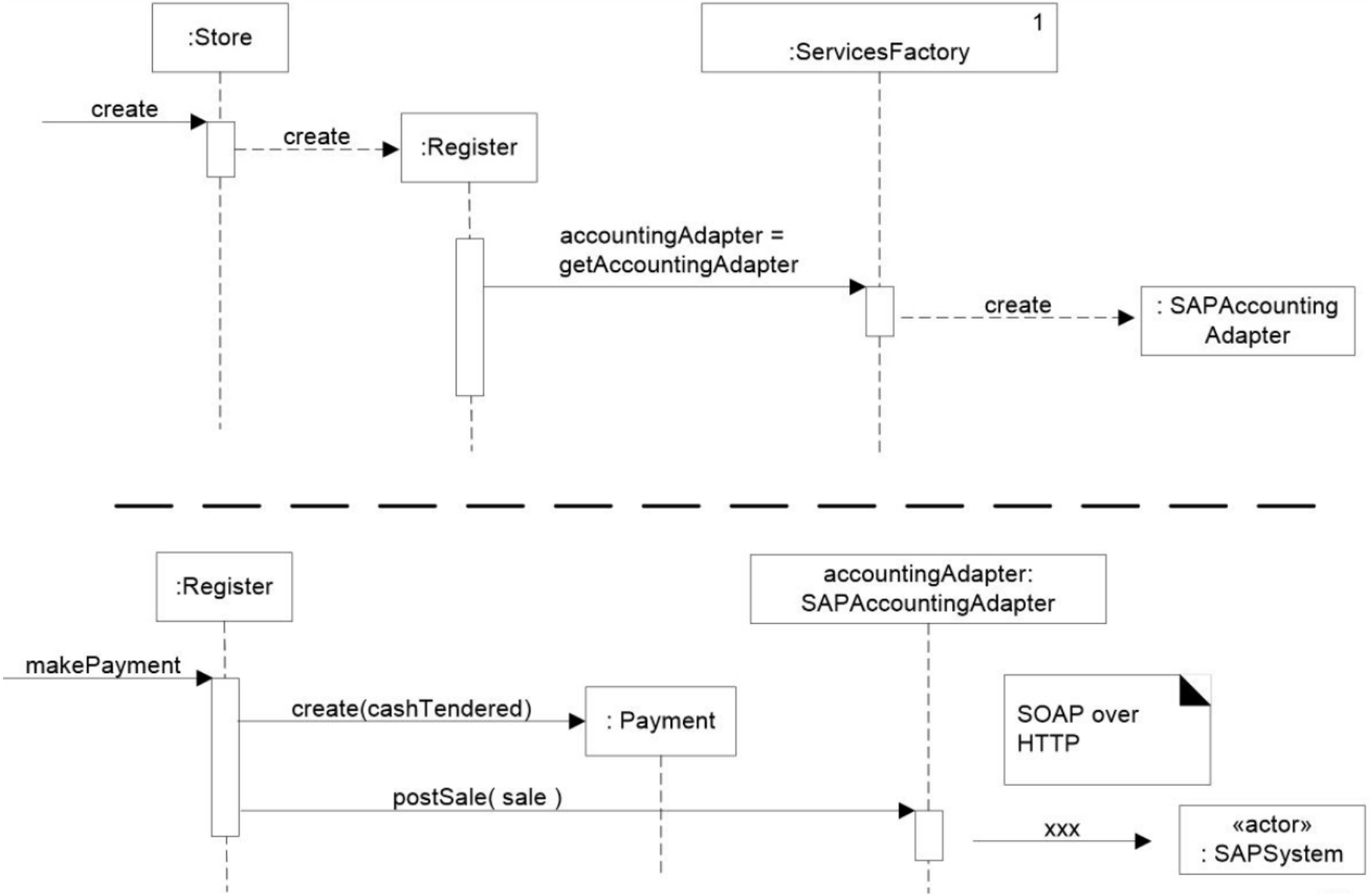


Servicios externos con diversas interfaces

- Combinación de Adapter, Factory y Singleton para proporcionar Variaciones Protegidas a las diferentes interfaces de servicios externos



Servicios externos con diversas interfaces



Diseño p/Requisito

- Siguiente problema de diseño a resolver: Proporcionar una lógica más compleja para la fijación de precios, como descuentos para un día en toda la tienda descuentos para personas mayores, etc.

- **Caso de Uso: Procesar Venta**
- La estrategia de fijación de precios (Regla, política, …) 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??

Patrón Strategy

- **Problema:** Cómo diseñar diversos algoritmos o políticas que estén relacionados? Cómo diseñar que estos algoritmos o políticas puedan cambiar?

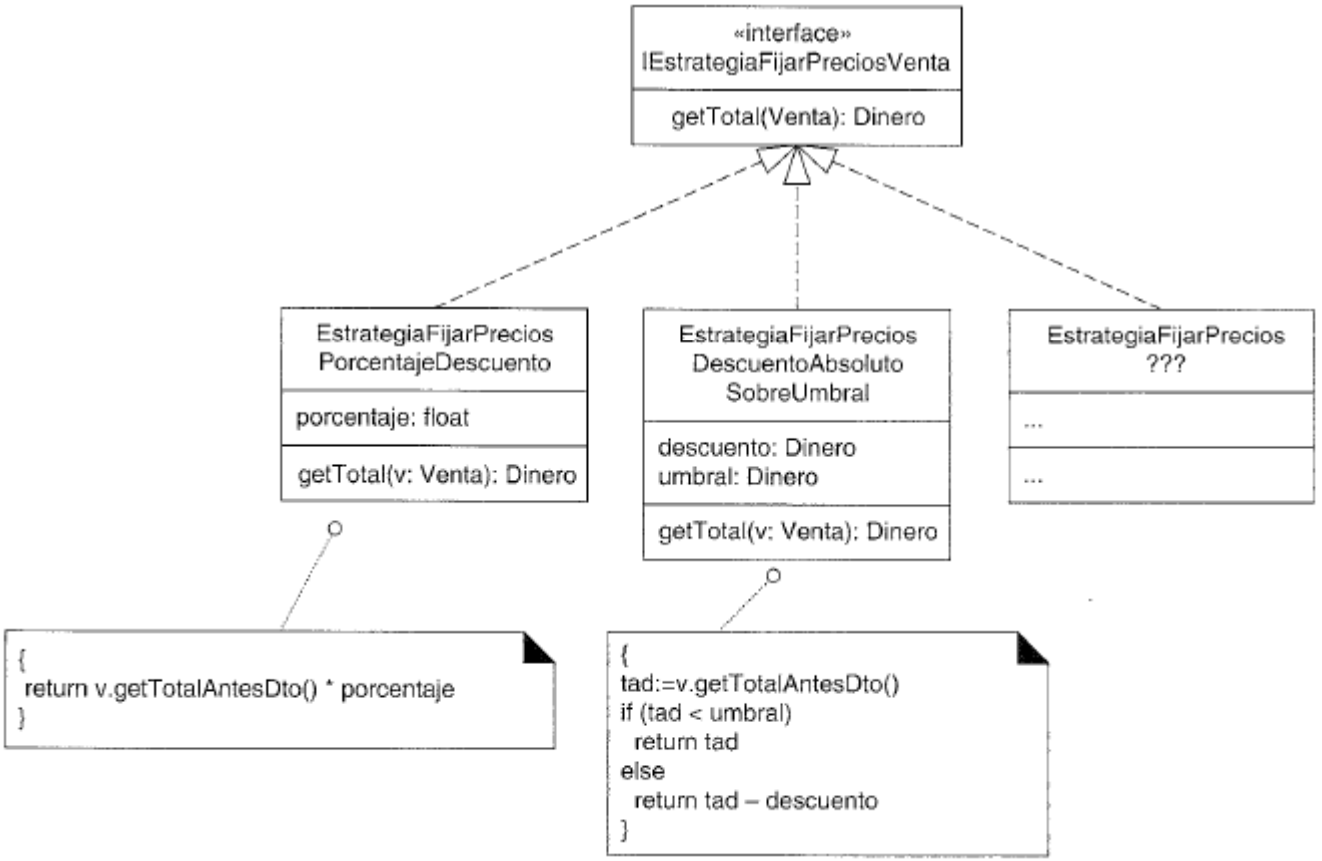
- **Solución**: Defina cada algoritmo/política/estrategia en una clase independiente, con una interfaz común.

Patrón Strategy

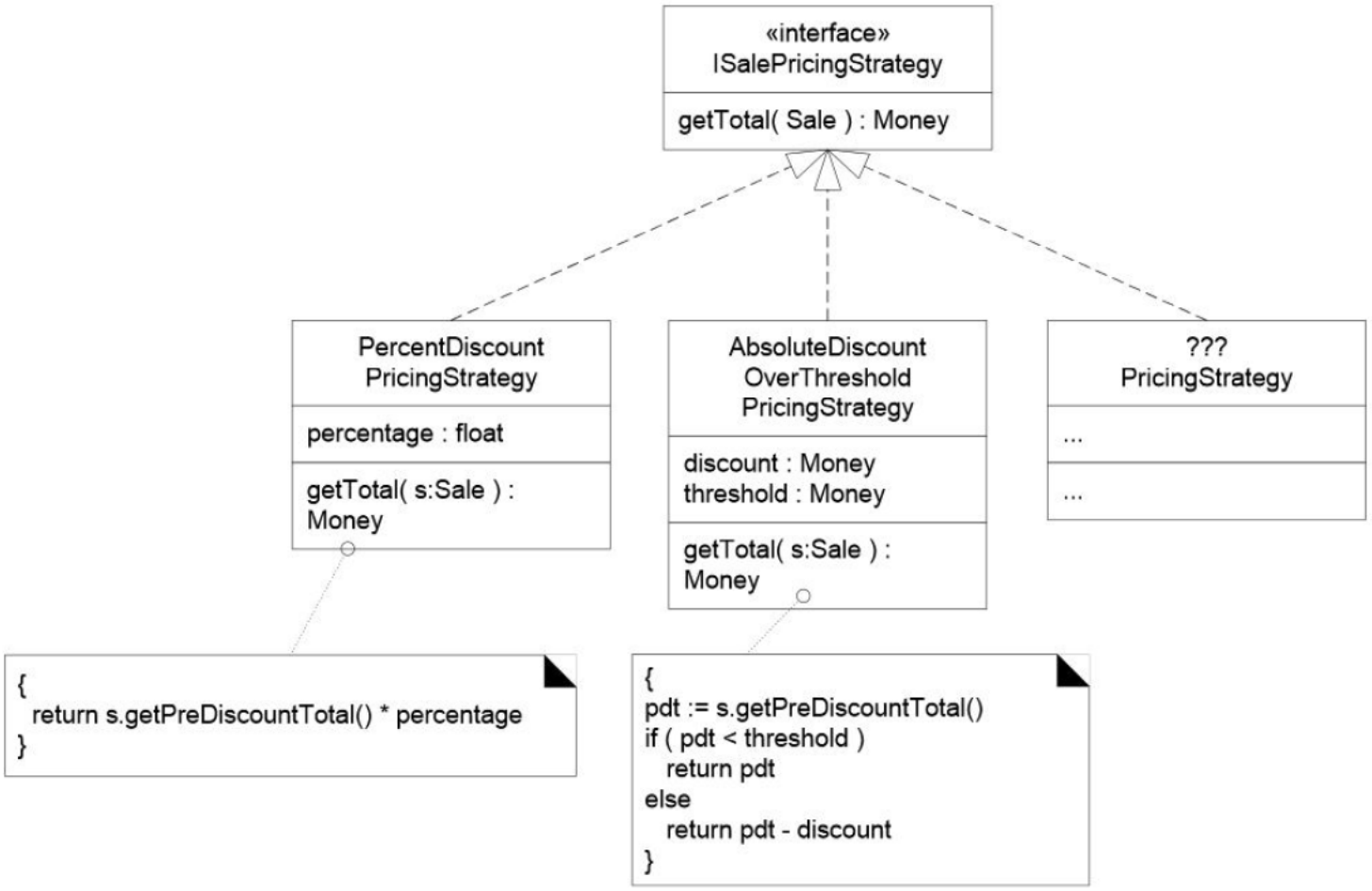
- El comportamiento de fijación de precios varía según la estrategia
- Se crean múltiples clases EstrategiaFijarPrecioVenta,
- Cada tipo de Estrategia se crea con un método polimórfico getTotal
- A cada método getTotal se le pasa como parámetro el objeto Venta, de manera que el objeto *estrategia pueda* encontrar el precio anterior al descuento y aplicar la regla.
- La implementación de cada getTotal será diferente en cada clase de Estrategia

Patrón Strategy.

Clases para la Estrategia de Fijación de precios

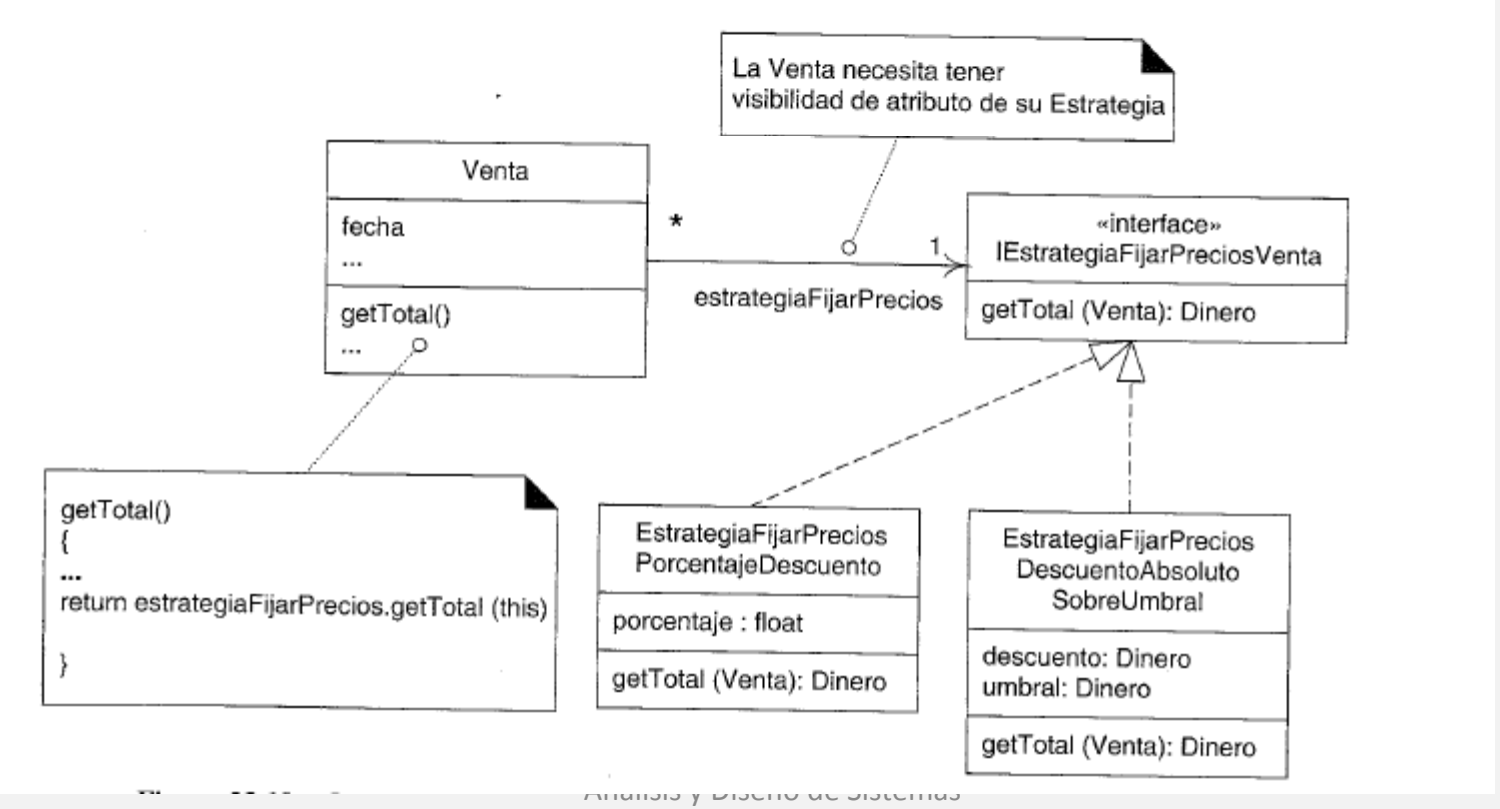


Patrón Strategy



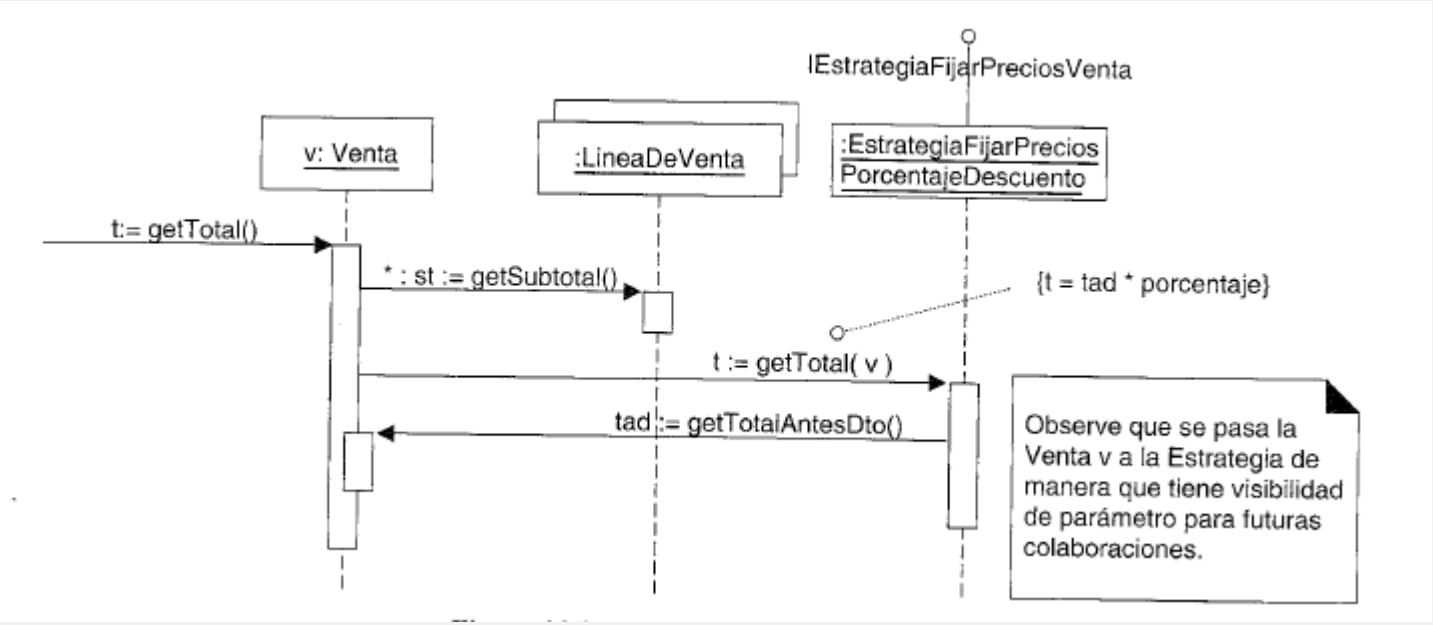
Patrón Strategy

- Un objeto *estategia* se conecta a un *objeto de contexto* (el objeto al cual se aplica el algoritmo)*, Venta* en este caso.
- El objeto de contexto necesita tener visibilidad de atributo de su estrategia.



Patrón Strategy

- Cuando se envía el mensaje getTotal a la Venta, delega parte del trabajo a su objeto estrategia
- Es habitual que el objeto de contexto pase una referencia a él mismo (this) al objeto estrategia
- No es necesario que el mensaje que se envía al objeto de contexto y al objeto estrategia tengan el mismo nombre.



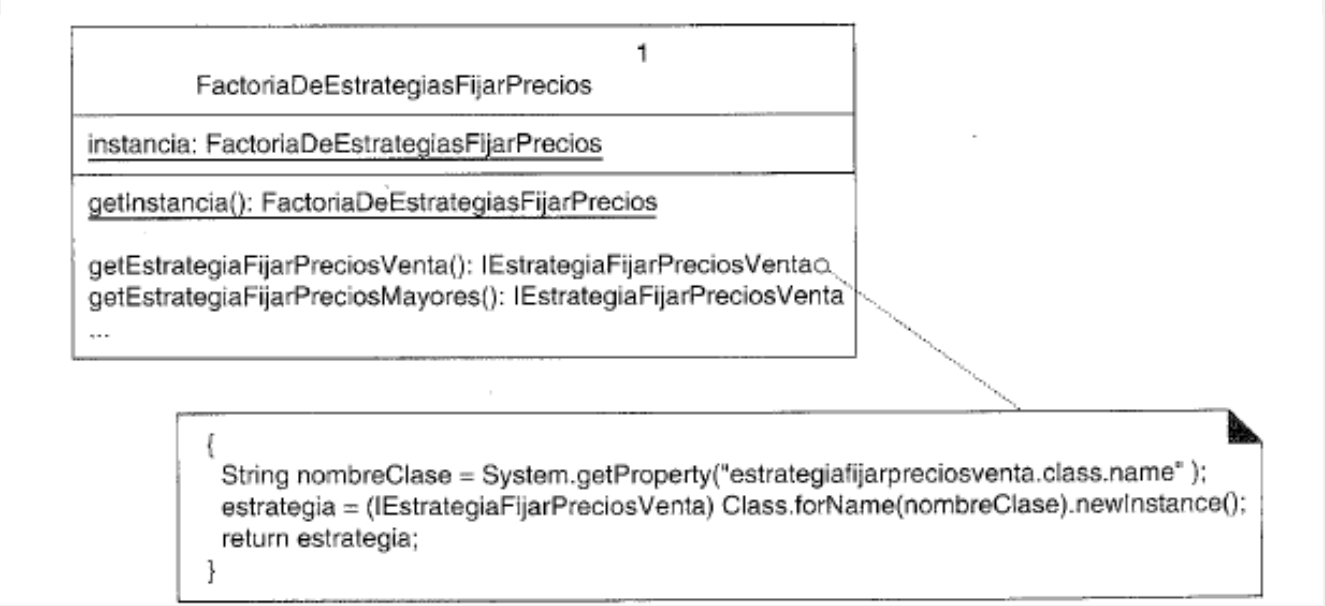
Creación de una Estrategia con una Factoría

- Existen diferentes algoritmos o estrategias de fijación de precios y cambian con el tiempo.

- **Quién debería crea la estrategia?**
- **Enfoque directo:** Aplicar el patrón Factoría
  - Una FactoríaDeEstrategiasFijarPrecios puede ser responsable de crear todas las estrategias que se necesitan en la aplicación
  - Se puede leer el nombre de la clase concreta de estrategia de fijación de precios como propiedad del sistema o de fuente de datos externa y crear una instancia.
  - Así se puede cambiar dinámicamente la estrategia.

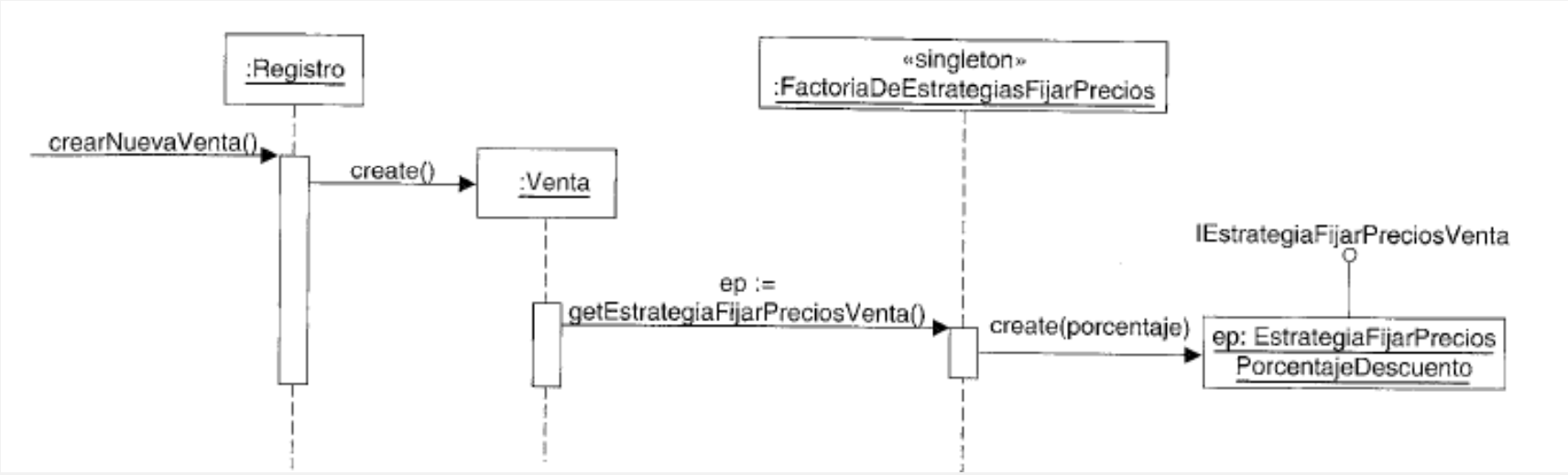
Factoría de Estrategias

- Como en la mayoría de las factorías, será un singleton y se accederá mediante el patrón Singleton.
- Debido a los cambios frecuentes, no se guarda la instancia de estrategia en un campo de la Factoría, sino que se crea cada vez que se necesita leyendo la propiedad externa.



Factoría de Estrategias

- Cuando se crea una instancia de la Venta, puede pedir a la factoría su estrategia de fijación de precios.



Lectura del Porcentaje

- Problema de diseño: Cómo encontrar los diferentes valores de los porcentajes o descuentos absolutos (que varían continuamente)?
  - Se almacenan en un almacén de datos externo, como una BD relacional, para cambiarlos fácilmente.
  - La Factoría los leerá y los asignará a la estrategia
  - El análisis de los puntos de variación y evolución con respecto al almacén de datos revelará si es necesario que se proteja contra variaciones.

Síntesis Aplicación de Strategy

- Con los Patrones Strategy y Factory se ha conseguido Variaciones Protegidas con respecto a las políticas para fijar precios que varían dinámicamente.
- Strategy se fundamenta en el Polimorfismo y las interfaces para permitir algoritmos conectables.

- **Patrones relacionados**:
  - ✓ Strategy se basa en el Polimorfismo y proporciona Variaciones protegidas.
  - ✓ Las estrategias normalmente se crean mediante una Factoría

Requerimientos y diseño

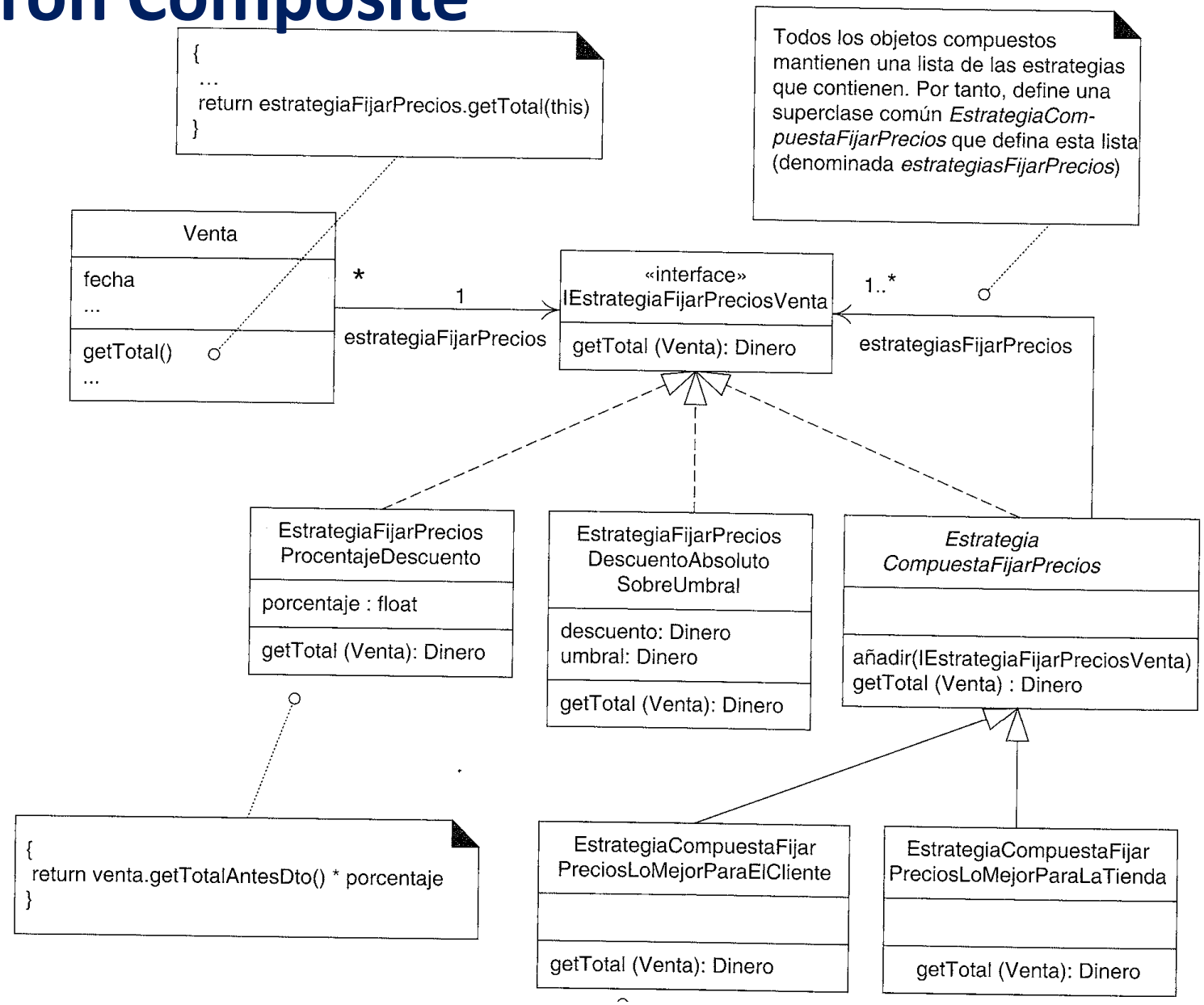
- **Problema de diseño**:
- Cómo se gestiona el caso de varias políticas contradictorias de fijación de precios? Por ej la Tienda tiene los jueves:
  - ✓ 20% de descuento a mayores
  - ✓ 15% de descuento en compras superiores a 300$
  - ✓ Los días jueves hay 50% de descuento en compras superiores a 600$Comprando una caja de té xxx, se obtien el 10% de descuento en el total
- En este caso, las estrategias de fijación de precios dependen de 3 factores:
  - ✓ Periodo de tiempo
  - ✓ Tipo de cliente
  - ✓ Un producto concreto

Requerimientos y diseño

- **Problema de diseño**:
- Se requiere que se defina la estrategia de resolución de conflictos de la tienda (normalmente “lo mejor para el cliente”)Pero esto podría cambiar (por ej. “el precio más alto”)
- Pueden existir múltiples estrategias para fijar el precio
  - ✓ Una estrategia puede requerir conocer el tipo de cliente y otra el tipo de producto, lo cual influye en el momento de su creación
- De qué forma se puede cambiar el diseño de manera que el objeto Venta no conozca si está tratando con una o más estrategias y ofrecer así un diseño para la Resolución de conflictos?

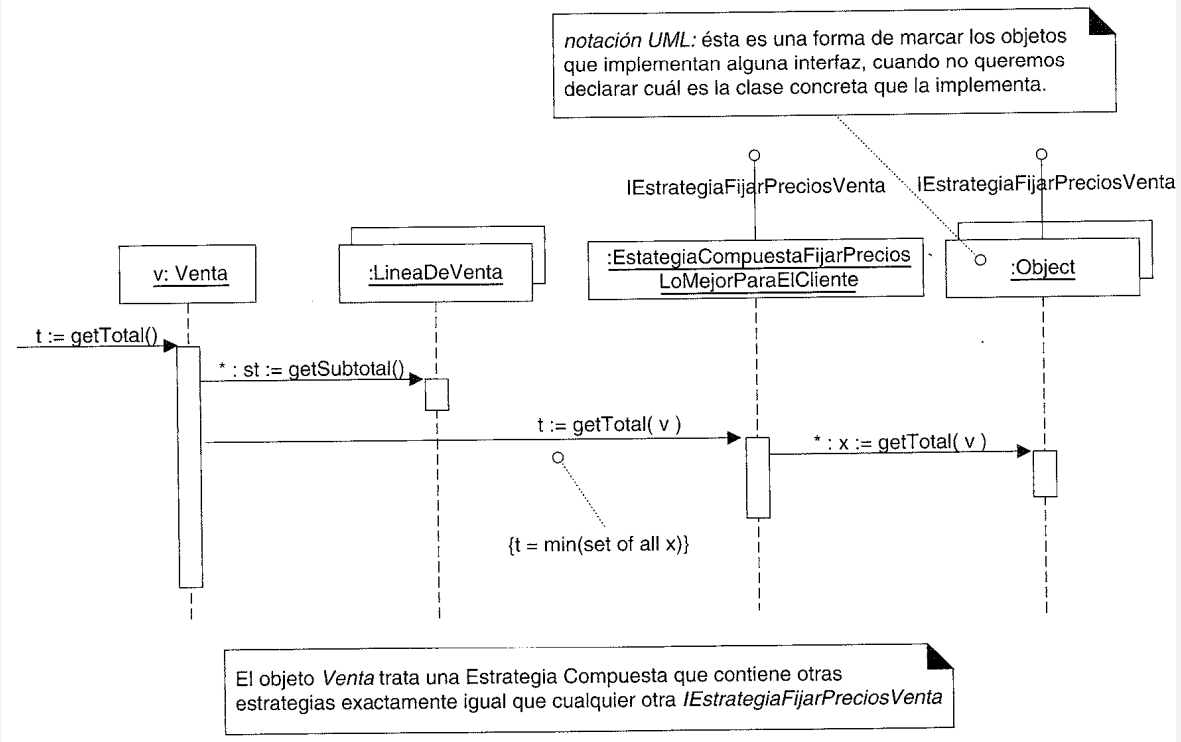
El Patrón Composite

Patrón Composite

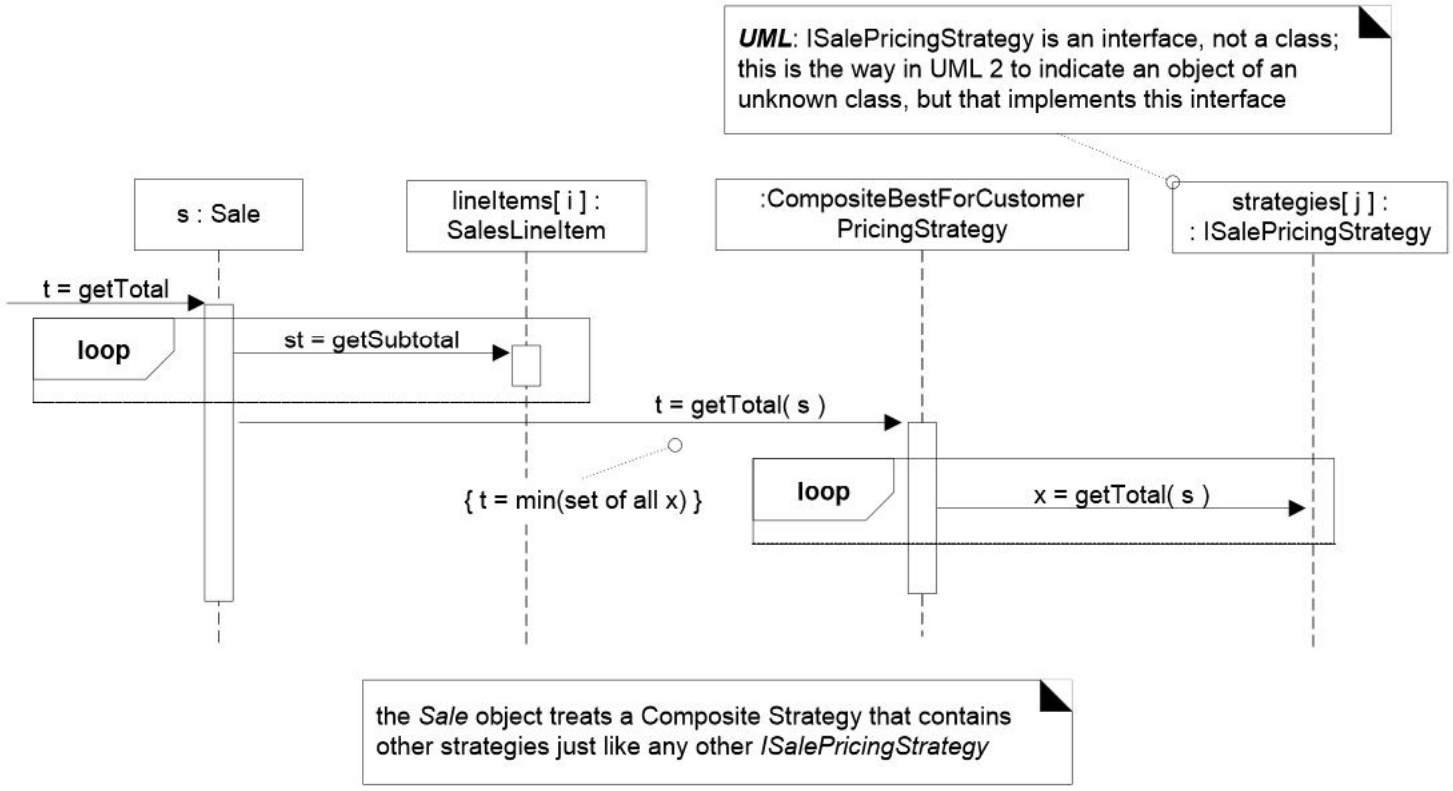


Patrón Composite

- La clase compuesta *EstrategiaCompuestaFijarPreciosLoMejorParaElCliente* hereda el atributo estrategiasFijarPrecios que contiene una lista de más objetos *IEstrategiasFijarPreciosVenta.* Se puede conectar el objeto Venta tanto a un objeto compuesto como aun objeto atómico.



Patrón Composite



Patrón Composite. Implementación

```
//superclase luego todas las subclases heredan una List de estrategias

public abstract class EstrategiaCompuestaFijarPrecios
   implements IEstrategiaFijarPreciosVenta
{

protected List estrategiasFijarPrecios = new ArrayList();

public añadir(IEstrategiaFijarPreciosVenta e)
{
   estrategiasFijarPrecios.add(e);
}

public abstract Dinero getTotal( Venta v );

}//final de la clase
```

Patrón Composite. Implementación

```
//Una Estrategia Compuesta que devuelve el total más pequeño
//de sus instancias EstrategiaFijarPreciosVenta internas

public class EstrategiaCompuestaFijarPreciosLoMejorParaElCliente
   extends EstrategiaCompuestaFijarPrecios
{

public Dinero getTotal(Venta venta)
{
   Dinero menorTotal = new Dinero (Integer.MAX_VALUE);

   //iteramos sobre todas las estrategias internas

   for( Iterator i = estrategiasFijarPrecios.iterator(); i.hasNext();)
   {
      IEstrategiaFijarPreciosVenta estrategia =
                          (IEstrategiaFijarPreciosVenta)i.next();
      Dinero total = estrategia.getTotal(venta);
      menorTotal = total.min(menorTotal);
   }
return menorTotal();
}

}//final de la clase
```

Creación de múltiples estrategias

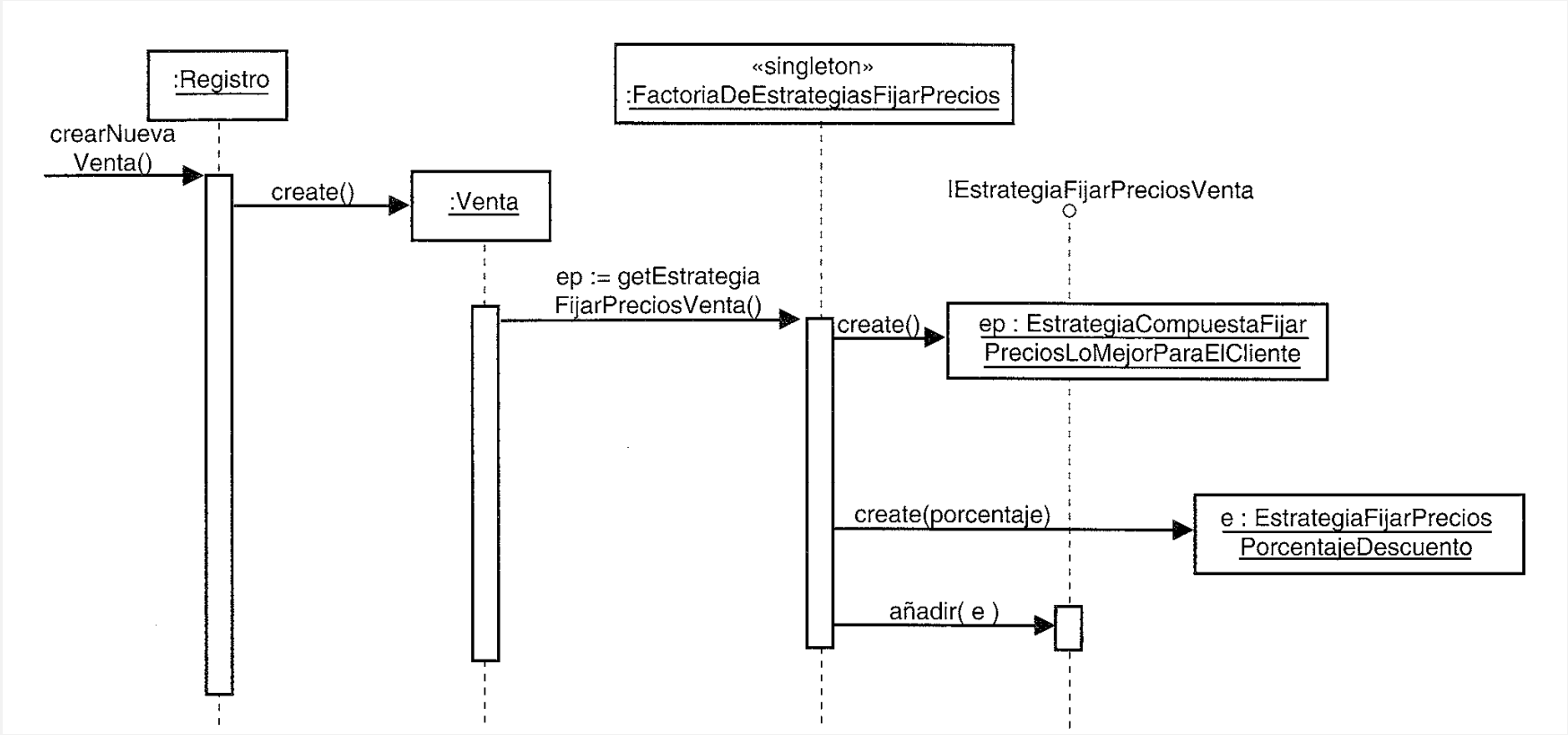
- Cuándo creamos las estrategias que implementan la interfaz IEstrategiaFijarPreciosVenta?
- El diseño comienza creando un Composite que contenga las políticas de descuento de la Tienda en el momento actual (alguna subclase de la jerarquía)
- Si en un paso posterior se descubre que se debe aplicar una nueva estrategia,se añade al objetoCompuesto, con el método heredado:
  EstrategiaCompuestaFijarPrecios.*añadir*

Creación de múltiples estrategias

- Puntos en el escenario donde se podrían agregar al objeto compuesto las estrategias de fijación de precios:
  - ✓ Descuento actual a nivel de tienda, se añade cuando se crea la venta
  - ✓ Descuento por el tipo de cliente, se añade cuando se informa al PDV del tipo de cliente
  - ✓ Descuento según el Producto, se añade cuando se introduce el producto en la Venta.

Creación de múltiples estrategias

- Diseño para el caso: Descuento actual a nivel de tienda, se añade cuando se crea la venta



Creación de múltiples estrategias

- Diseño para el caso: Descuento por el tipo de cliente
- Se recuerda la extensión del Caso de Uso que identificó este requisito

Caso de uso UC1: Procesar Venta

…

Extensiones (o Flujos Alternativos):

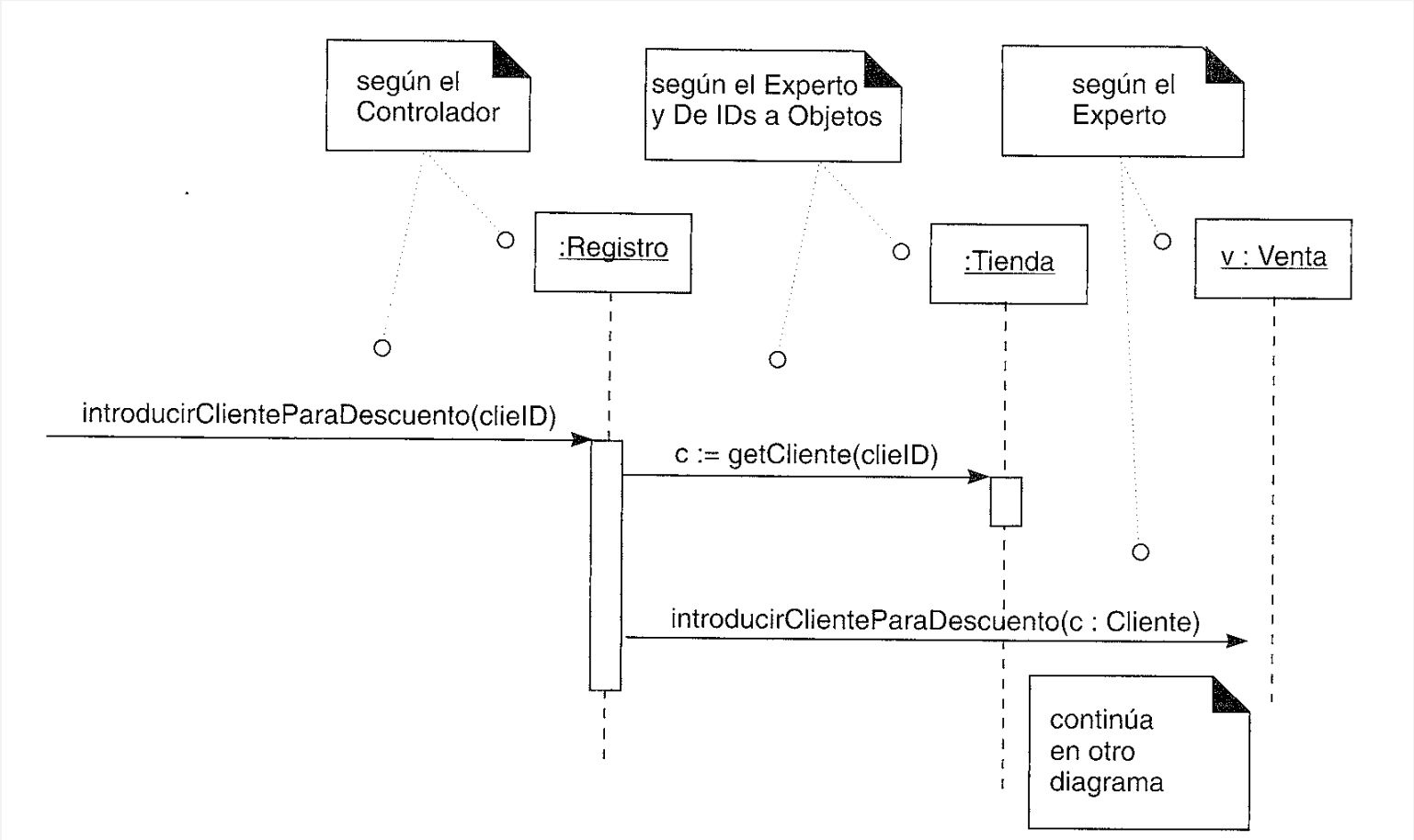
5b. El Cliente dice que le son aplicables descuentos (ej. empleado, cliente preferente):

1. El Cajero señala la petición de descuento.
2. El Cajero introduce la identificación del Cliente.
3. El Sistema presenta el descuento total, basado en las reglas de descuento.

- Aparece una nueva operación del sistema: introducirClienteParaDescuento

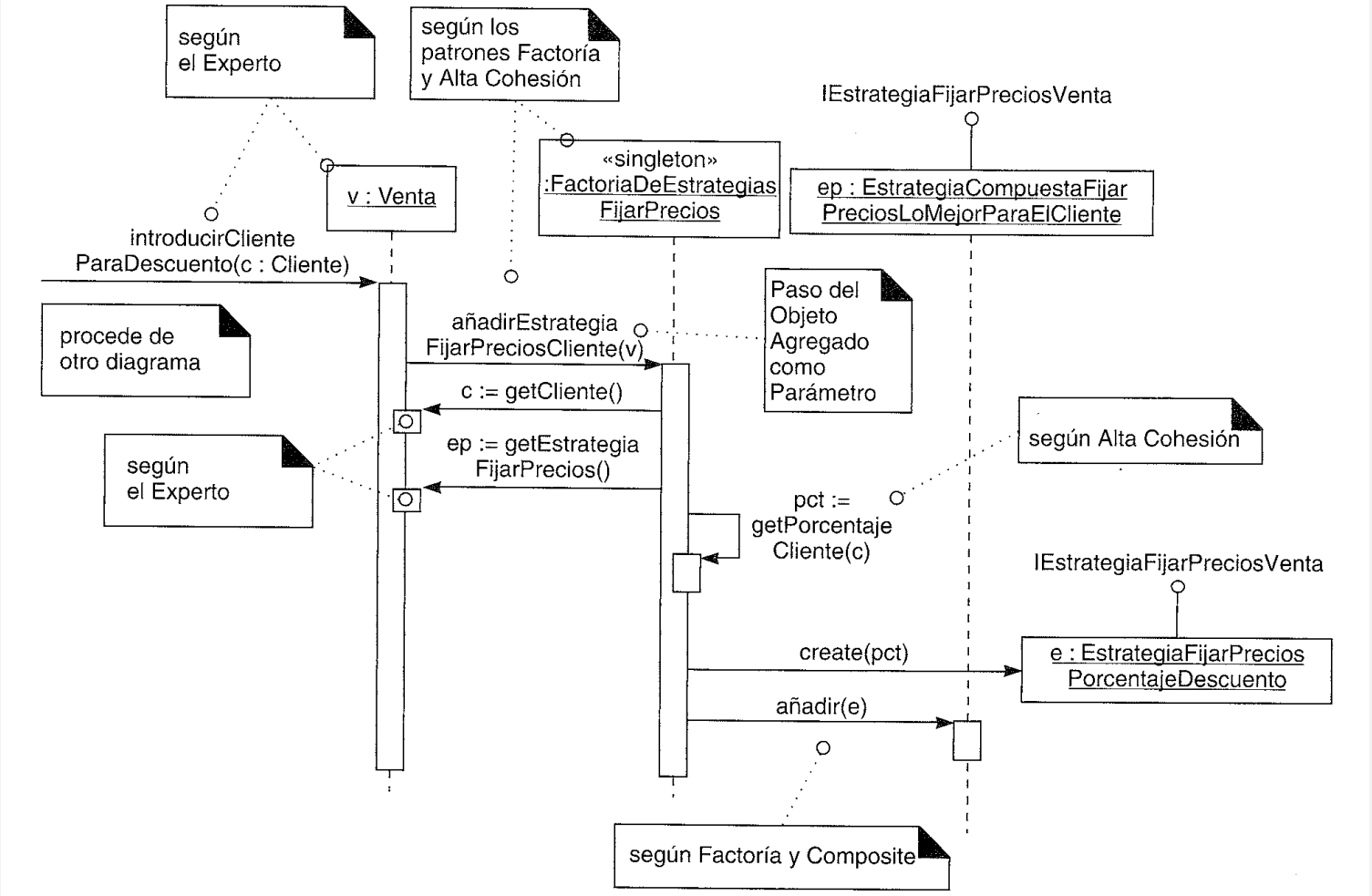
Creación de múltiples estrategias

- Diseño para el caso: Descuento por el tipo de cliente (Parte 1)
- Es preciso introducir la nueva operación de sistema



Creación de múltiples estrategias

- Diseño para el caso: Descuento por el tipo de cliente (Parte 2)
- Es preciso introducir la nueva operación de sistema



Patrón Composite.

- Patrones relacionados: El Patrón Composite se utiliza normalmente con los patrones Strategy y Command.

- Composite se basa en el Polimorfismo y proporciona Variaciones Protegidas a los clientes de manera que no les afecta si el objeto con que se relacionan es atómico o compuesto.

Bibliografía

- UML y Patrones, Larman, 2004
- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma E, Helm R. Johnson R. y Vlissides J. Addison Wesley, 1995-2005
- Head First Design Patterns. Freeman E & E. Ed. O'Reilly, 2004