Diseño de interfaces entre componentes

Estudio del grado y forma de acoplamiento entre partes de un sistema

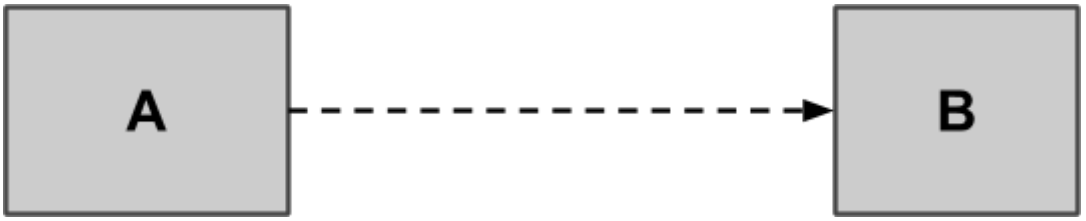
por
Fernando Dodino
Nicolás Passerini
Franco Bulgarelli

Versión 2.5
Abril 2017

Índice

1 Interacción entre dos componentes

Volvemos una vez más a decir que un sistema es un conjunto de componentes que se relacionan para lograr un objetivo común. Ahora bien, ¿cómo se relacionan esos componentes?



Supongamos que A "necesita algo" de B. Lo que nos va a interesar es ver qué alternativas tengo para modelar esa línea punteada.

Algunas cosas a tener en cuenta:

- Dirección y sentido de la comunicación:
  - si la comunicación va de A hacia B (control directo) o al revés (inversión de control)
  - si la comunicación entre A y B es bidireccional o unidireccional
- Si A y B están ambos en el mismo ambiente
- Manejo de errores
  - qué pasa si hay un error de conexión con B[1]
  - qué pasa si hay error cuando B procesa el pedido de A
- Grado de dependencia:
  - si A esperará activamente a que B termine de procesar (es decir, si la comunicación es sincrónica)[2] o puede continuar trabajando sin bloquearse (comunicación asincrónica)
  - si A necesita algún resultado de B o si simplemente debe notificarlo y olvidarse (*fire and forget*)
- determinar si interesa que B procese el pedido de A en forma exitosa para que A haga otras cosas en la misma transacción
- determinar si se necesita reprocesar (en forma manual o automática) determinados mensajes de A a B
- determinar si se necesita obtener un log (llevar un registro) de los pedidos enviados a B
- etc.

Las decisiones que tomemos impactarán en **el grado y forma de acoplamiento entre los componentes de nuestro sistema**. Por eso, a continuación trataremos varias de estas cuestiones.

[1] Leer también apunte de manejo de errores
[2] Leer también apunte de comunicación entre componentes

1.1 Digresión: ¿por qué componente?

Las ideas que estamos mencionando aplican no sólo cuando componente = objeto. Podemos mencionar diferentes niveles para diseñar componentes, en una clasificación que no es taxativa, sino que intenta ordenar un poco nuestros pensamientos:

- Micro diseño, nivel de “programa”, pienso en objetos, clases, procedimientos, funciones
- Nivel medio, muchos objetos forman un componente/módulo/package y pienso la integración de esas cosas
- Nivel arquitectura de software, el nivel más alto dentro de mi aplicación, veo los elementos principales de mi aplicación y cómo se integran entre sí.
- Nivel "arquitectura de sistema", pensando el sistema como un conjunto de piezas de software o integración entre sistemas. Pienso cómo se integran los sistemas / aplicaciones entre sí.

Los patrones que veremos a continuación son aplicables a cualquiera de estos niveles, más allá de que los componentes que elijamos en nuestros ejemplos sean pequeños por cuestiones didácticas.

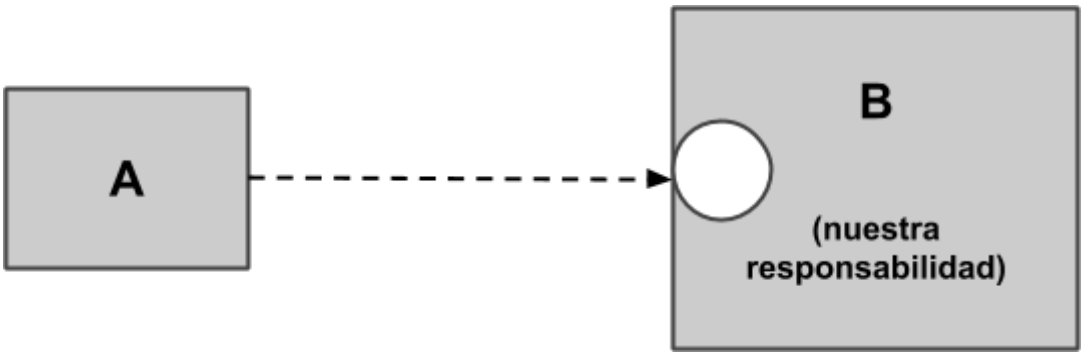
2 Interfaces entrantes y salientes

Nos encontramos diseñando un componente. Primero, respondamos una pregunta fácil respecto al problema de la comunicación: ¿estamos diseñando la forma de que otros componentes "hablen" con el nuestro, o la forma de conectarnos a otro componente? Es decir, ¿estamos diseñando una interfaz entrante o saliente?

Analicemos ambos casos.

> Cuidado: es importante notar que la tecnología en la que están implementados los componentes que se comunicarán con el nuestro podría ser muy diferente a aquella con la que trabajamos.

2.1 Interfaz entrante



Cuando estamos en esta situación, tenemos que pensar cual es la información que nuestro componente (B, en el ejemplo) necesita para resolver su tarea, y cómo nos conviene recibirla. Tendremos que considerar que las decisiones que tomemos impactarán en cuan fácil es para otros miembros de un equipo de desarrollo usar este componente.

2.2 Interfaz saliente

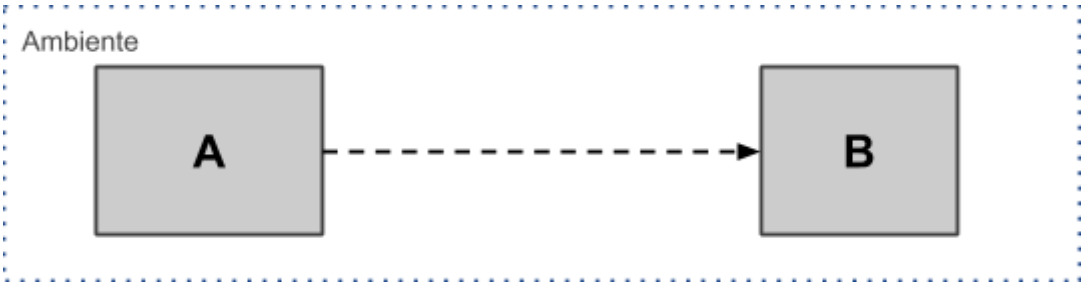
Ahora nos encontramos en la situación contraria: estamos desarrollando un componente que necesita delegar en otro ciertas tareas. Ahora tenemos que preocuparnos por proveer la información que éste (en el ejemplo, B) necesita, adaptándonos a la forma que nos propone de comunicación.

3 Ambientes

Debemos entender que el diseño de la comunicación entre componentes es bastante diferente cuando los componentes están en el mismo o diferente ambiente.

3.1 Todo en un solo lugar

A y B pueden ser módulos que conviven en el mismo ambiente.



Aquí no tenemos mayores restricciones en cuanto a la comunicación. Si estamos trabajando con objetos esto es tan sencillo como enviar un mensaje a un objeto y ya.

A y B se piensan como componentes que tienen una separación lógica entre sí:

- Porque A o B pueden estar desarrollados por otros (objetos de la JDK, de frameworks de terceras partes, de grupos de trabajo que no forman parte de la misma gerencia a la que pertenecemos, etc.)
- Aun cuando A y B son componentes que nosotros mismos desarrollamos, tratamos de reducir su acoplamiento. Ojo, los componentes tienen relación entre sí y está muy bien que la tengan; no quiero eliminar la interacción, sí evitar que A conozca demasiado de B o viceversa. Entonces puedo pensar una interfaz que me permita cambiar implementaciones posibles para B:
    - porque no tengo implementado B pero necesito ir testeando A, entonces armo una implementación mínima de B que responda al mensaje de A para luego reemplazarla por la implementación real de B
    - porque B no tiene el alcance cerrado y quiero dejar la puerta abierta para los cambios que se que van a venir
    - o por cualquier otra razón que lo justifique

3.2 Distintos ambientes

En el segundo caso, A y B están implementados en distintos lugares (componente remoto), distintas tecnologías o incluso desarrollados bajo diferentes paradigmas.

Si bien es imposible independizarse completamente de cuestiones como la tecnología o la ubicación de estos componentes[3] (¿están la misma computadora? ¿O debemos accederlos a través de la red de Internet?), intentaremos que esto nos afecte lo menos posible.

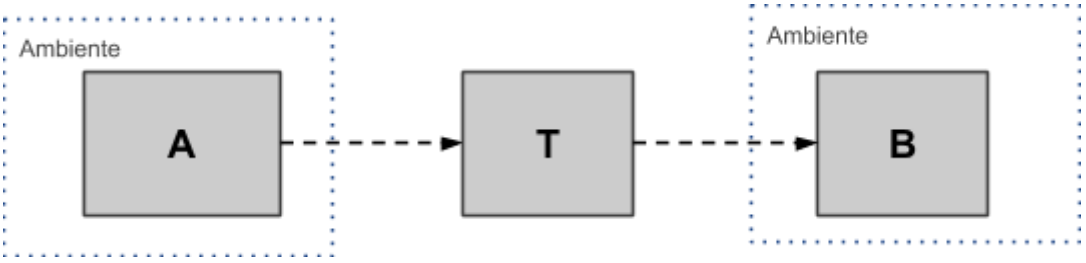
¿Por qué? Porque nos interesa minimizar el acoplamiento entre los componentes. Así, si por ejemplo estamos desarrollando componentes bajo el paradigma de objetos, nos interesará diseñar las interfaces bajo este paradigma.

[3] Los requerimientos no funcionales, al igual que el diablo, están en los detalles

Si para poder hacer que los componentes se comuniquen con algo muy diferente, será necesaria una transformación, pero no será parte de mi componente, sino que será un elemento independiente que sí dependerá de la tecnología, pero que debería estar desacoplado de la interfaz.

Entonces el objetivo es separar

- la lógica de “comunicación” a nivel negocio: qué quiero decir, qué información debo pasar, qué espero recibir, qué errores de negocio podrían ocurrir
- de la lógica de “transformación” que fuera necesaria.



T es el componente que hace algún tipo de transformación. Puede estar en alguno de los dos ambientes, o en otro ambiente aparte. No es objetivo de la materia desarrollar el componente T.

4 Interfaces sincrónicas y asincrónicas

Otra pregunta a responder es: cuando el componente A se comunica con B, ¿en qué grado necesita A los resultados de B? Tenemos dos opciones: comunicaciones sincrónicas y asincrónicas.

4.1 Interfaces sincrónicas

Se da cuando un componente le envía un mensaje a otro, éste se queda esperando a que el otro termine. Esta es la forma en que trabaja el envío de mensajes en objetos, la ejecución de procedimientos en imperativo, o la aplicación de funciones en funcional.

Es también la forma más simple de trabajar, porque es fácil razonar sobre ella: si un componente A le envía un mensaje a otro B, y luego A le envío otro mensaje a C, sabemos con seguridad que la respuesta de C llegará después de la respuesta de B.

Un ejemplo: nos interesa conocer la cotización del dólar de la fecha de hoy. Si nos paramos desde el componente A y pensamos en hablar con un componente B que obtiene la cotización, una opción sincrónica es la siguiente:

```
BigDecimal dolarHoy = cotizador.obtenerCotizacion()
```

Lo interesante del sincronismo es que sabemos que si la línea termina de ejecutarse sin errores, tendremos con total seguridad un resultado, que usarlo en el siguiente envío de mensajes:

```
BigDecimal precioFinal =
     calculadoraPrecios.obtenerPrecioFinal(dolarHoy)
```

La contra de esta forma de comunicarnos es que si la comunicación entre A y B es lenta o poco confiable[4], detendrá la ejecución del componente A por quizás demasiado tiempo.

4.2 Interfaces asincrónicas

Cuando A está en condiciones de continuar su trabajo mientras B calcula el resultado, o incluso, cuando a A no le importa el resultado de B, podemos implementar interfaces asincrónicas.

En este caso, A le enviará un mensaje a B para solicitar el inicio de la tarea, con los parámetros que necesite, y luego continuará su flujo de ejecución. Y en tanto B, cuando tenga el resultado la tarea listo, de alguna forma comunicará a A de este suceso. Algunas formas de hacerlo son[5]:

- depositar el resultado de la operación en algún lugar de memoria compartida, el cual A periódicamente revisa
- guardar una referencia a A, y enviar un mensaje a éste cuando haya terminado
- guardar un callback

Ejemplo: podemos construir una interfaz asincrónica para nuestro cotizador de la siguiente forma:

1. En un primer momento el componente A dispara el mensaje `cotizador.obtenerCotizacion()`
2. ... pasa el tiempo, y el cotizador (componente B) pone la respuesta en el cotizador `cotizador.setUltimaCotizacion(cotizacion)`
3. Más tarde, el componente A irá a buscar la respuesta a cotizador.ultimaCotizacion

O también podríamos usar callbacks (bloques de código):

```
cotizador.obtenerCotizacion([ resultado | ..usar resultado... ])
```

Y luego B se encargará de ejecutar el callback cuando sea necesario.

[4] Lo cual requerirá muchas veces reintentar la operación
[5] Ver el apunte de patrones de comunicación entre componentes

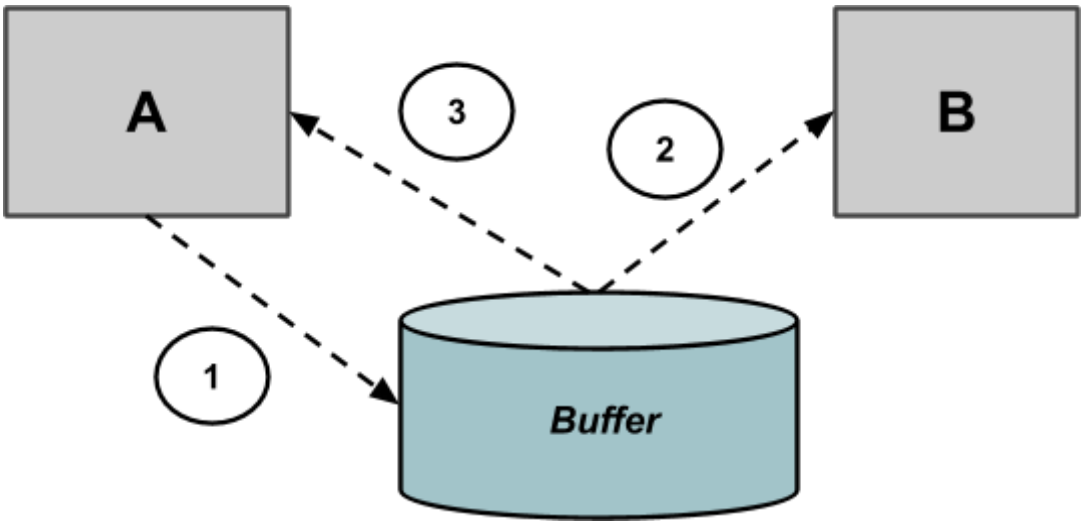
Como casos extremos, podría darse que a A no le importara que B realmente le responda algo, sino tan solo que se vea notificado. En este caso, A sólo se debe enviarle un mensaje a B con la información que necesite, sin preocuparse por capturar un resultado.

Una interfaz sincrónica que no expone un resultado, es más fácilmente convertible a una interfaz asincrónica que una que sí expone un resultado.

4.2.1 Buffers

Muchas veces, para poder lidiar con el asincronismo y el hecho de que A no procesará instantáneamente el resultado de B, o que B no iniciará instantáneamente la tarea solicitada por A, necesitaremos un buffer que almacene temporalmente el resultado y/o parámetros de la tarea.

Si bien este buffer puede ser una parte del componente B y no merece un estudio aparte, otras veces este buffer es externo, y tanto A como B deberán empezar a hablar con buffer, en lugar de entre ellos.



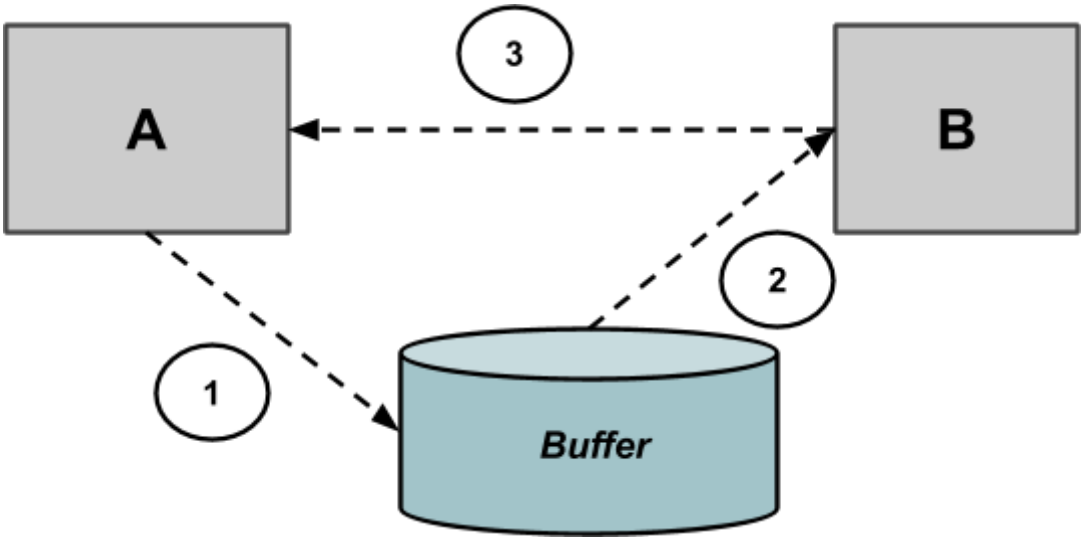
1. *A* envía el mensaje al buffer (en lugar de enviárselo a *B*)
2. Un proceso toma la información del buffer,
    a. envía un mensaje a *B*
    b. y la respuesta de ese mensaje se envía a A.

Desde la perspectiva de A, necesitamos una interfaz entrante para recibir el resultado, con lo cual lo que antes era un caso de uso ahora son dos:

- el pedido de una acción a B
- el procesamiento de la respuesta de B ante ese pedido, donde se acepta o rechaza la transacción.

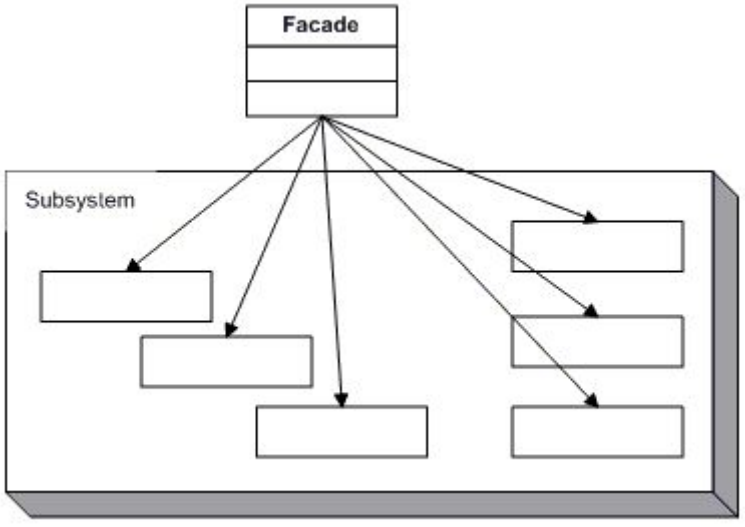
Desde la perspectiva de B tenemos varias opciones:

- si el buffer tiene cierta inteligencia, entonces B no se ve afectado por el mecanismo sincrónico / asincrónico de la interfaz: sigue siendo una interfaz entrante y no hay muchos cambios en la implementación.
- si por el contrario B toma la responsabilidad de avisar el resultado a A, la interfaz entrante disparará a su vez una interfaz saliente con resultado ok o error una vez procesado el pedido. Esto se ve en la figura siguiente:



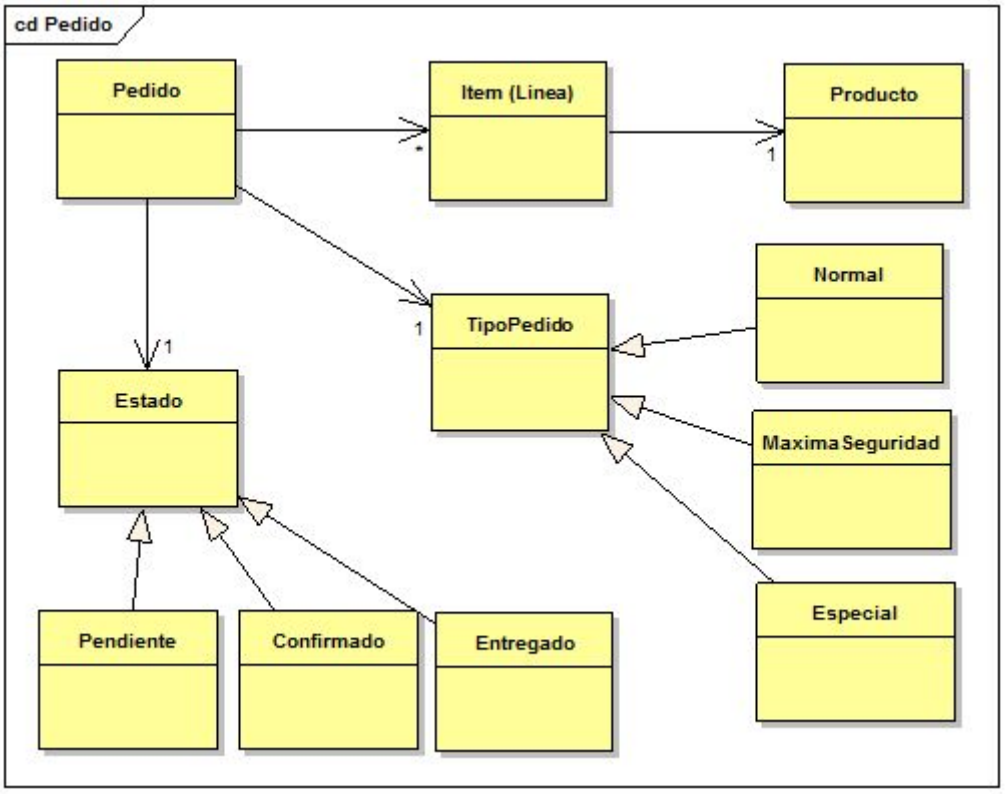
5 Algunas palabras sobre las Fachadas (Façades)

Existe un patrón conocido como Façade cuya motivación es proveer una abstracción que permite ver a un conjunto de objetos como si fueran uno solo.



Por eso se suele utilizar en algunos casos para modelar interfaces entrantes.

Ejemplo: tenemos un sistema de pedidos. Cada pedido tiene líneas o ítems (*x* cantidad de producto *P*), un criterio que define la urgencia y cuidado de los productos que conforman el pedido y un estado que define qué hacer ante determinadas acciones.



Para generar un pedido,

a. quiero agregar 5 kilos de queso parmesano, 2 kilos de dulce de batata
b. también quiero asignarle el estado (por ejemplo, default en pendiente)
c. y setearle el tipo de pedido.

Si pensamos métodos en Pedido para facilitar estas tareas:

a. para agregar un ítem

```
public void addItem(int cantidad, Producto producto) {
      this.items.add(new Item(cantidad, producto);
}
```

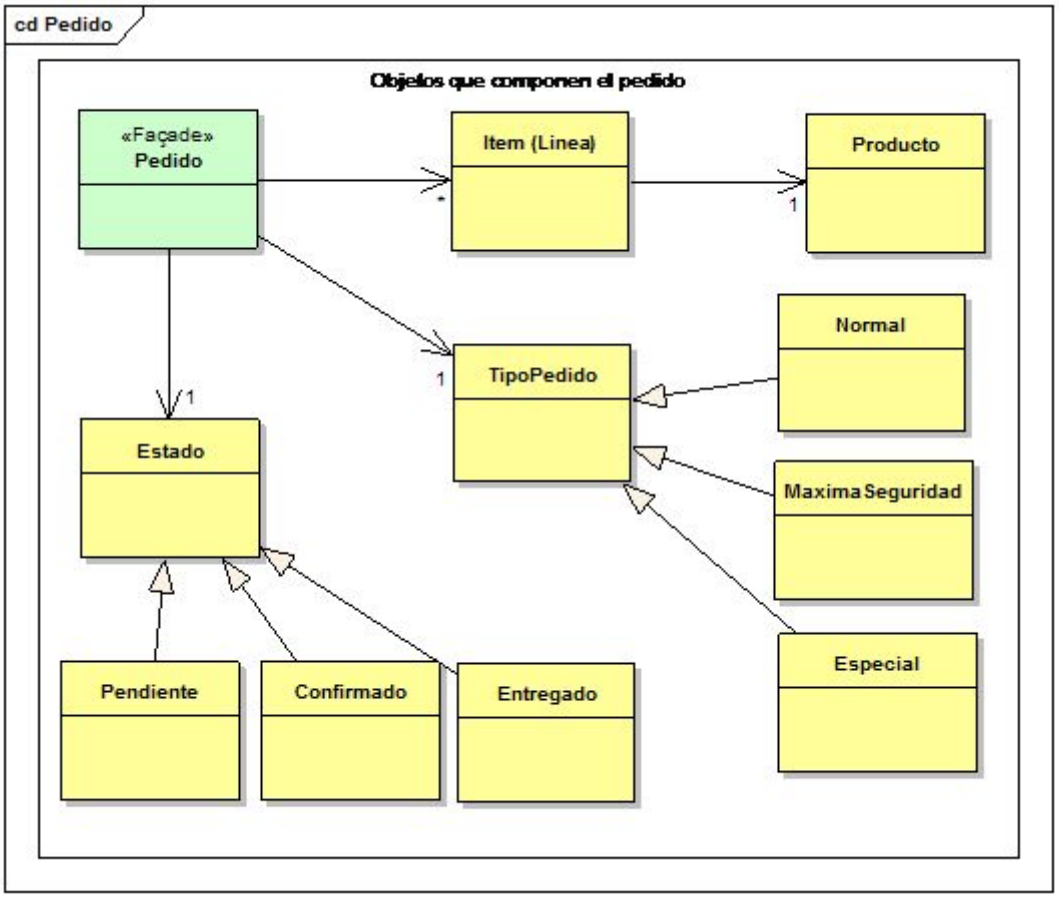
b. para asignar estado default:

```
public Pedido() {
      ...
      this.estado = new Pendiente();
}
```

c. para setear el tipo de pedido

```
public void setTipoPedidoMaximaSeguridad() {
      this.tipoPedido = new MaximaSeguridad();
}
```

Entonces el Pedido está actuando como un Façade: el que genera un pedido nuevo no necesita conocer los ítems, ni las subclases de estado ni las subclases del tipo de pedido.

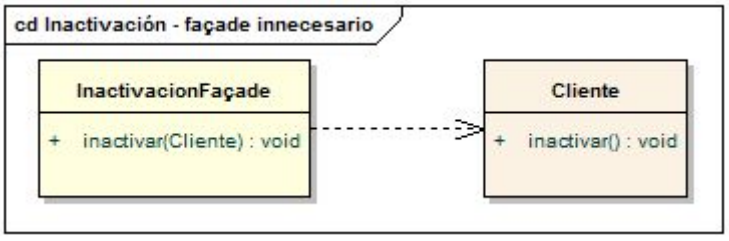


El Façade me permite tener un punto de acceso desde el cual ofrecer servicios a componentes externos. Eso permite a los demás hablar con un único componente nuestro y elimina la necesidad de conocer nuestro modelo.

5.1 Advertencias sobre el Façade

No obstante no es necesario tener siempre un objeto que oficie de fachada, **en especial si lo único que hace es redirigir el mensaje al objeto de negocio que hace el trabajo que corresponde**.

Ejemplo: en la inactivación aparece un objeto InactivacionFaçade



¿Qué hace el método inactivar de InactivacionFaçade?

```
public void inactivar(Cliente cliente) {
      cliente.inactivar();
}
```

No agrega ningún valor, solamente forwardea el pedido al cliente.

Además, si el objetivo del Façade es ver un conjunto de objetos como si fueran uno solo eso no se está cumpliendo: estamos recibiendo un objeto cliente como parámetro

```
public void inactivar(Cliente cliente) {
      cliente.inactivar();
}
```

Entonces si el que invoca a InactivacionFaçade tiene que obtener de alguna manera al cliente, la pregunta es: ¿por qué no le envía el mensaje inactivar() directamente al cliente?

Corolario: solamente usar Façade cuando usamos un objeto que simplifica la interacción con un módulo de un sistema que puede ser bastante complejo (como en el caso de Pedido que permite centralizar las acciones sobre los demás objetos que manejan su estado).

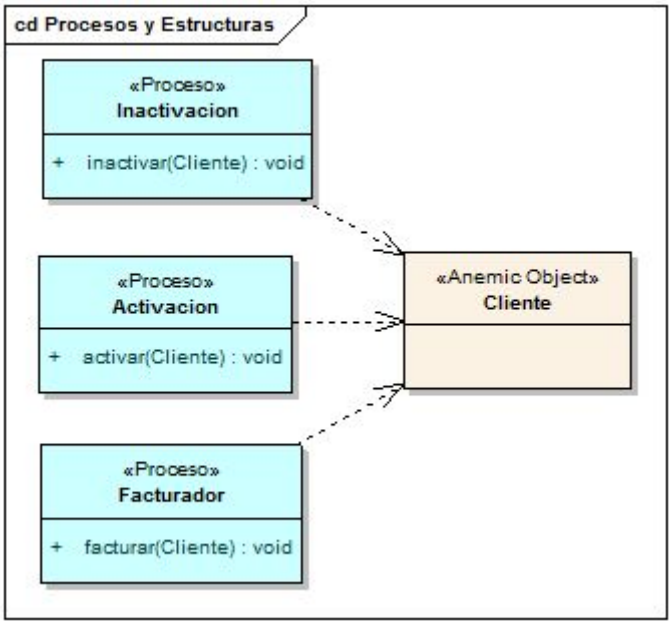
Corolario 2: En el común de los casos la **interfaz entrante se modela con un mensaje a un objeto de negocio y con eso nos alcanza**.

5.2 Malos usos del Façade

Suele ser un error común del diseñador novato confundir al Façade con un Manager, que quiere suplantar al objeto de dominio tomando decisiones que no son propias:

```
>>InactivarFaçade
public void inactivar(Cliente cliente) {
      cliente.setEstado("I");
}
```

Si el Façade toma decisiones que deberían estar en cliente (o en producto o en documento de compra), es señal de que estamos yendo hacia un mal diseño: el cliente se vuelve una estructura de datos pasiva sobre la cual otros objetos modifican su estado interno.



La inactivación en sí misma no es un God Object, pero sí la suma de los objetos estereotipados como <<procesos>>. El resultado es un diseño pobre que no se parece en nada a las ideas de objetos: cualquier cambio en la estructura de Cliente repercute en todos los objetos que hacen las acciones.

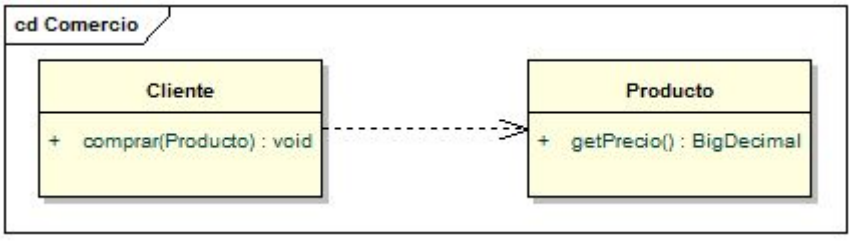
6 Ejercicio práctico: compra con tarjeta de crédito

Ahora veremos cómo modelar una interfaz a través de un ejemplo concreto

Tenemos un sistema de ventas de un comercio cualquiera que tiene cuenta corriente con sus clientes. Necesitamos que cada venta que se realice con tarjeta de crédito notifique al sistema Credit System informando:

- Número de tarjeta de crédito
- Monto de la venta
- Descripción de la operación
- Comercio origen
- Fecha y hora

Tenemos un bosquejo del diagrama de clases:



No nos importa cómo se termina de resolver el método comprar, sólo nos interesa agregar una responsabilidad más: disparar la interfaz saliente contra el sistema de Tarjeta de Crédito.

6.1 Interfaz saliente

Dos cosas al respecto:

1. Está bueno llevar esa responsabilidad a un objeto aparte
2. Por otra parte, aun si quisiera poner toda la responsabilidad en Cliente, ¿cómo me comunico con Credit System? No sabemos cómo implementar esa interfaz: el enunciado no dice nada al respecto.

Y en realidad no dice nada porque no es relevante para esta materia.

Podemos pensar muchas soluciones técnicas:

- RPC (Remote Procedure Call)
- Socket puro
- Web Service

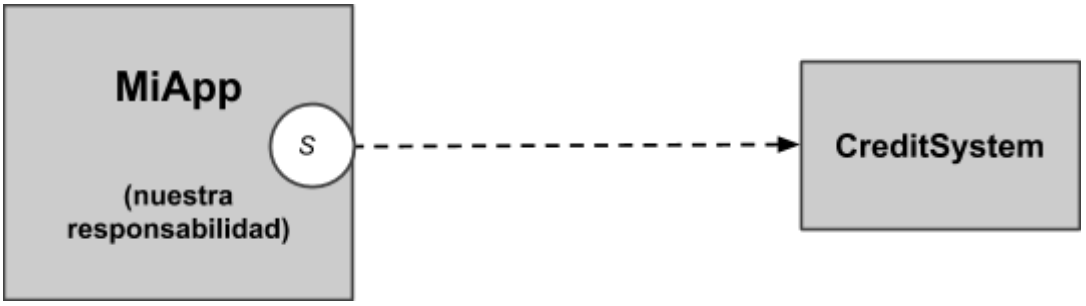
Cualquiera sea la tecnología que elijamos, podemos abstraer la idea de “notificar la compra de un cliente que tiene tarjeta de crédito”.

Sabemos que una abstracción puede implementarse en objetos de muchas maneras:

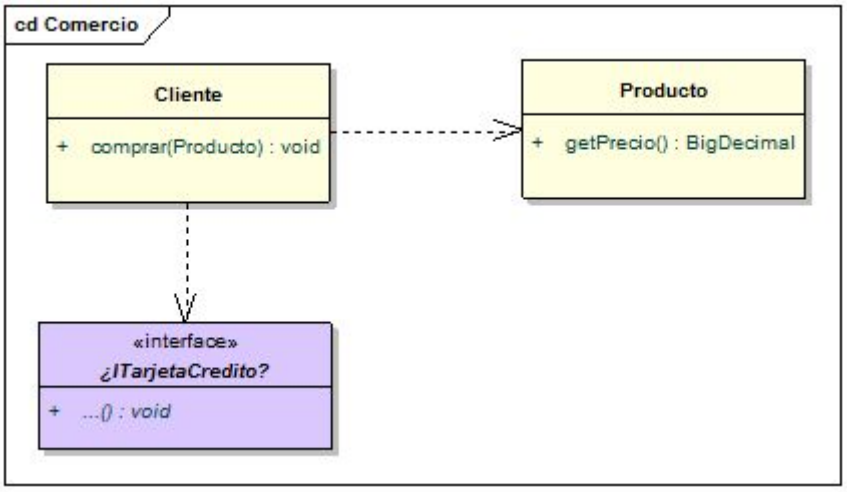
- Con un objeto
- Con una clase
- Con una interfaz (la de Java)
- Con un método
- Con un mensaje

etc.

En nuestro caso, podemos pensar en una interfaz de Java para modelar la notificación de la compra (interfaz saliente entre componente A, nuestra aplicación y componente B, una aplicación específica para Tarjetas de Crédito):



Este objeto S será de una clase que implemente la interfaz que estamos definiendo ahora. Lo importante es que yo no necesite cambiar la firma del método cuando tenga que implementar la comunicación posta.



Algunas opciones posibles para definir la interfaz:

a. `notificarCompra(Integer numeroTarjeta, BigDecimal monto, String operacion, String CUITComercioOrigen, Date fechaCompra)`

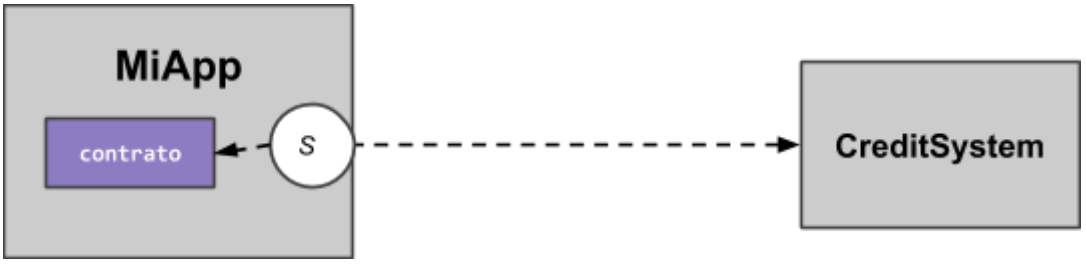
b. `notificarCompra(Cliente cliente, Producto producto, String operacion, String CUITComercioOrigen, Date fechaCompra)`

c. `notificarCompra(Compra compra)`

- a) soporta menos cambios en la interfaz que b): si en algún momento nos piden que enviemos la razón social del cliente eso nos fuerza a cambiar la firma del método de a).
- Tanto a como b proponen un contrato con ¡5 parámetros!, proponer una abstracción que relacione los elementos que intervienen en la compra es una técnica apropiada para reducir el impacto de cualquier cambio futuro (la heurística nos dice que hay altas chances de que un método que recibe 5 parámetros sufra alguna modificación en el futuro cercano). Esta abstracción puede incluso ser útil para el negocio, como en el caso de la Compra.

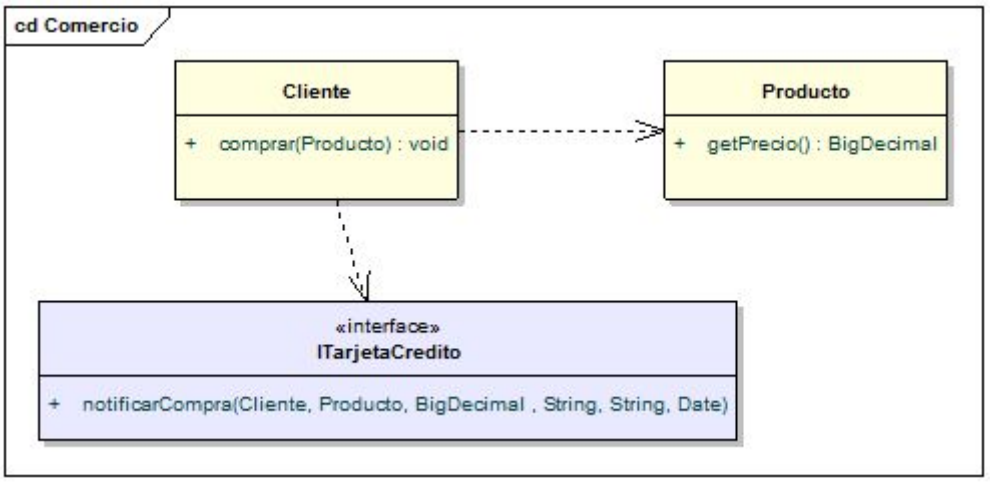
En definitiva, lo importante es:

- reconocer que necesitamos poner la responsabilidad en una interfaz y
- saber qué información va a necesitar el que termine implementando esa interfaz



El contrato (la definición del mensaje con los parámetros) es lo que se pide en la materia.

Y el diagrama de clases queda:



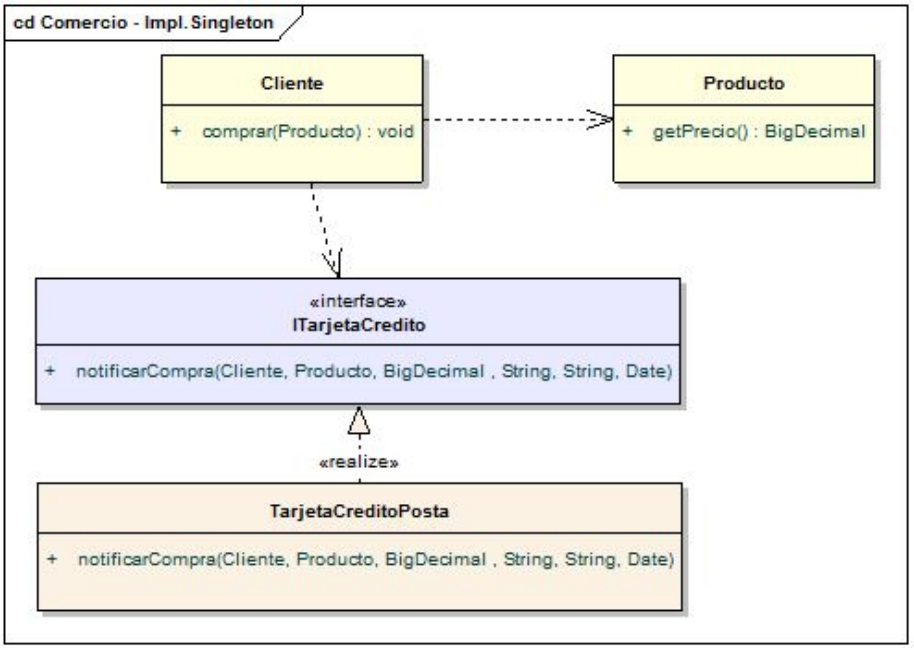
TODO: Cambiar la firma del método

sin necesidad de determinar la implementación de dicha interfaz.

6.1.1 Cómo queda la compra en el código

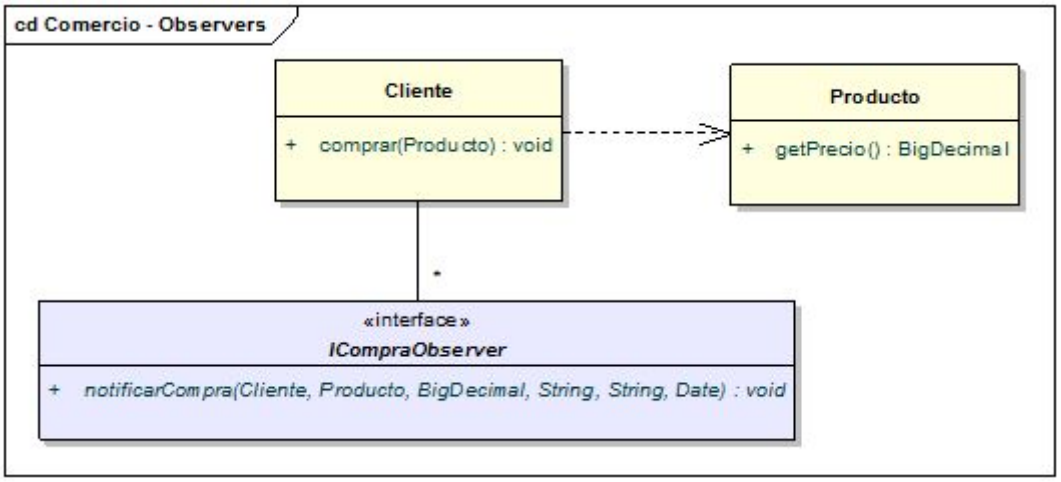
Una opción posible es acceder al sistema Credit System desde un único punto de acceso global **(Singleton)**.

```
>>Cliente
public void comprar(Producto producto) {
	...
	if (this.tieneTarjetaDeCredito()) {
		TarjetaCreditoPosta.getInstance().notificarCompra(this,
producto, "Compra xx", Comercio.CUIT_Origen, new Date());
	}
	...
}
```



Otra opción es tener conjunto de “interesados” (observers, listeners) que implementan la interfaz saliente:

```
>>Cliente
public void comprar(Producto producto) {
      ...
      for (ICompraObserver compraObserver :
this.comprasObservers) {
            compraObserver.notificarCompra(this, producto,
"Compra xx", Comercio.CUIT_Origen, new Date());
      }
      ...
}
```



Ambas opciones son válidas, aunque es importante recalcar que si nuestra intención es únicamente disparar una interfaz saliente y no nos interesa poder activar o desactivar esa interfaz, el Observer es un caso de sobrediseño: nuestra solución provee una complejidad mucho mayor al problema que quiere resolver. Por otra parte sí es correcto que pensemos en el Observer cuando en el evento que activa la notificación a esa interfaz saliente hay otros interesados o existe esa posibilidad.

6.1.2 Manejo de Errores en la interfaz

¿Qué sucede si tenemos que manejar errores al llamar a la interfaz?

Si seguimos pensando dentro del paradigma de objetos, los errores los manejamos con excepciones y no con códigos de retorno.

Entonces preferimos pensar en notificarCompra como void, y no como un int notificarCompra.

¿Qué errores podemos recibir?

- Errores en la conexión con el host remoto, error en la comunicación (ruido o corte de conexión), errores de permiso en el acceso, errores en el formato del mensaje, etc.
- Errores de programa al ejecutar el procesamiento del mensaje
- Errores de negocio: “El cliente no existe”, “El monto no puede superar los $ 1.000 por operación”, etc.

Los dos primeros son errores de sistema/programa. En el último caso tenemos errores de negocio. Dependiendo de cada caso quizás tengamos que contemplar acciones posibles o simplemente mostrar el mensaje de error en la pantalla (interfaz de usuario).

Si la tecnología que usamos para comunicar A con B no tiene mecanismo de excepciones y nos devuelve un código de error, es en definitiva problema de quien implemente la interfaz ITarjetaCredito... lo ideal es convertir ese código de error 30018 en una excepción que sea lanzada para quien pueda atraparla correctamente (en muy pocos casos un objeto de negocio y en la mayoría de las veces una ventana de usuario).

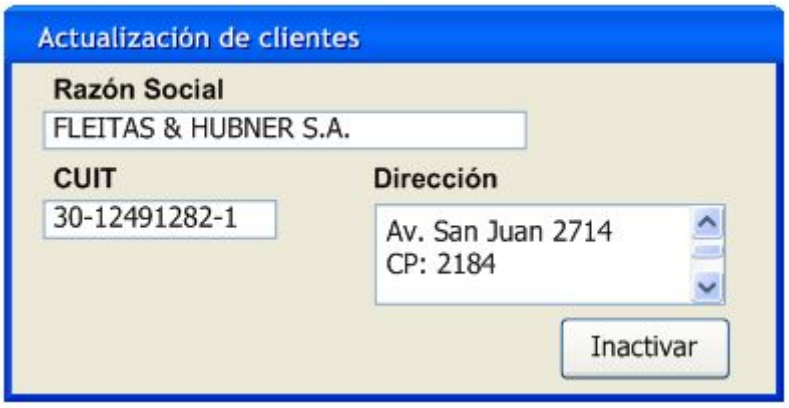
Entonces la implementación “posta” de la tarjeta de crédito actúa como un objeto “wrapper” que envuelve la llamada real (que retorna el int) y lo convierte en una excepción:

```
>>Implementación de la tarjeta de crédito
public void notificarCompra(...) {
      ...
      int codigoError = ...<conectarse mediante alguna tecnología con
el sistema de tarjeta de crédito> ...;

      if (codigoError == 30018) {
           throw new SystemException("... <formato del mensaje con
error> ...");
      }

      if (codigoError == 30015) {
         throw new BusinessException("El monto no puede superar los $
1.000 por operación");
      }

      ...
}
```

6.2 Interfaz entrante

En el mismo sistema queremos modelar la inactivación de un cliente, que puede dispararse:

a. Desde la interfaz de usuario de nuestra aplicación



b. Desde el sistema de análisis crediticio “Customer Checker”, un enlatado que no forma parte de nuestra aplicación

Los clientes inactivos no pueden hacer compras.

Estamos modelando ahora una interfaz entrante combinando dos escenarios posibles:

- Para el caso a): un módulo que reside en el mismo ambiente del negocio (o no, dependiendo de qué tecnología elijamos para desarrollar la interfaz de usuario)
- Para b): un componente externo

¿Quién es responsable de inactivar un cliente? El cliente, por supuesto, respondiendo al mensaje inactivar():

A los fines prácticos de este apunte no importa mucho

- cómo se codifica el método inactivar()
- qué validación hay que agregar al método comprar()

sabemos que habrá un impacto en esos métodos y nos alcanza con eso.

6.2.1 Cómo definir la interfaz entrante

Para modelar una interfaz entrante basta con definir qué objeto de negocio es el que responde a ese pedido: la tecnología hará la magia de convertir ese pedido en un mensaje para dicho objeto.

Nro.cliente	Fecha	Activar/Inactivar
102875	10/11/2009	1