Decisiones de diseño para construir aplicaciones

por Nicolás Passerini y Fernando Dodino
Técnicas Avanzadas de Programación

Versión borrador - Octubre 2009

Resumen

Una aplicación completa necesita complementar la visión del modelo de dominio de un requerimiento (pensando en los objetos, sus responsabilidades y sus relaciones con otros objetos) con otros temas o "concerns" (como la interfaz de usuario, la persistencia, la posibilidad de aceptar entornos distribuidos, la seguridad, etc.) El riesgo es que al integrar las tecnologías para construir una aplicación olvidemos todo lo aprendido sobre las cualidades que nuestro software debe tener. El objetivo del presente apunte es analizar las decisiones que tenemos que tomar (tanto en la arquitectura como a nivel diseño) para tener un sistema funcionando.

Índice

1. Introducción: División de Concerns **3**
1.1. Concerns y capas 4

2. Interfaz de usuario (Presentación) **6**
2.1. Elementos básicos 6
2.2. MVC 13
2.2.1. Extendiendo el MVC 17
2.3. Pasando en limpio 18

3. Aplicación **20**
3.1. Definición 20
3.2. Un ejemplo 20
3.2.1. Objetos caso de uso: ingreso de un socio 21
3.2.2. Objetos Proxys 24
3.2.3. Objetos DTO 25
3.2.4. Objetos DAO 28
3.3. Otras responsabilidades del caso de uso 30
3.4. Extensiones al Caso de Uso 32
3.5. Límites 36

A. Videoclub: Enunciado **38**
A.1. Actualizar clientes 38
A.1.1. Pantalla principal 38
A.2. Alquilar Películas 40

1. Introducción: División de Concerns

Cuando tenemos que modelar un requerimiento como:

"Calcular el descuento que se aplica a un cliente de un banco que pide un préstamo..."

nuestro foco está puesto en encontrar abstracciones que representan lo que comúnmente decimos es el "negocio", el lenguaje que domina el usuario:

- el objeto cliente representa al cliente real de un banco
- el mensaje `getDescuento()` abstrae la responsabilidad del sistema de determinar el descuento y se la asignamos al objeto cliente

Ahora bien, las aplicaciones tienen muchas cosas tecnológicas de las que tenemos que ocuparnos, además del negocio:

- **Interfaz de usuario o presentación**: la tecnología a utilizar para ofrecer una interfaz amigable al usuario
- **Persistencia**: cómo almaceno y cómo recupero la información más allá del ambiente donde creo los objetos
- **Aplicación**: qué objetos son los intermediarios entre los objetos de dominio y los demás concerns (principalmente persistencia y presentación). Dentro de este concern aparecen
    - **los objetos de acceso a datos**: son los que saben cómo recuperar a los clientes morosos o las facturas pendientes (abstraen las diferentes consultas que se hacen al repositorio de objetos persistentes).

- **los objetos que modelan un caso de uso**: conocen todas las tareas necesarias para llevar a cabo un requerimiento del usuario. Ejemplo: en el alta de un cliente hay que abrir una transacción, instanciar a un cliente y persistirlo en la base.

- **Manejo de transacciones**: la posibilidad de definir transacciones de manera de asegurarnos que varias operaciones ocurren todas o ninguna a la vez es algo deseable en una aplicación.
- **Manejo de la concurrencia a la información**: el acceso simultáneo a un objeto por dos procesos (el objeto puede o no persistirse, en el primer caso este concern se relaciona con la persistencia)
- **Distribución**: cómo interactúan objetos que están en distintos ambientes.
- **Seguridad**: la forma de registrarse en la aplicación, el manejo de perfiles y autorización para las distintas acciones del sistema, el encriptado y desencriptado de datos, etc.
- **Trace y Log**: obtener un detalle de los errores que surgen, auditar las operaciones que hace el usuario, detectar cuellos de botella, administrar o configurar el sistema "en caliente" (al mismo tiempo que se sigue ejecutando), obtener información de debug, monitorear el uso de memoria, etc.

  Entre otros temas...

Estos forman parte de requerimientos no funcionales, no por eso menos importantes.

1.1. Concerns y capas

El término concern suena parecido a capa, no obstante creemos oportuno aclarar que la división de tareas o cosas de las que me tengo que ocupar no necesariamente implica la división física de ambientes o máquinas. Más adelante discutiremos la necesidad de tener ambientes distribuidos, o ambientes diferenciados (un application server o VM donde vivan los objetos de aplicación y de negocio, otro donde estén los objetos de presentación, etc.) y el análisis de los costos y beneficios de ir por este tipo de soluciones. Por eso preferimos trabajar con cada concern como **decisiones de diseño que hay que tomar para construir una aplicación**.

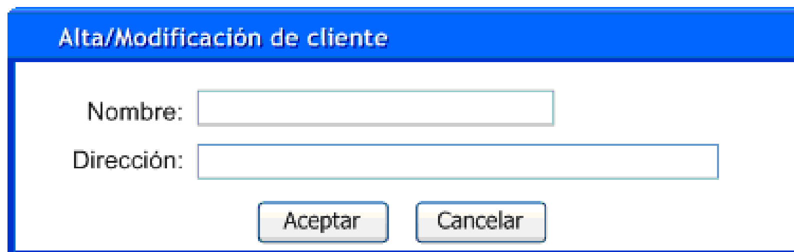
Nuestro objetivo A continuación vamos a dar una base teórica para estudiar distintas alternativas al modelar una aplicación en los concerns de presentación, aplicación y persistencia y permitiremos al lector un estudio comparativo que servirá para que sus aplicaciones respeten las cualidades deseadas de todo diseño. Para ello tomaremos como requerimiento base una aplicación para un videoclub (ver Anexo).

2. Interfaz de usuario (Presentación)

Definimos interfaz de usuario a todo lo que le permite al usuario interactuar con el sistema [2]. Si bien uno asocia la parte de presentación a una pantalla, también podríamos incluir cualquier tipo de dispositivo tecnológico (sensores con displays, cajeros automáticos, etc.) Sólo a los fines de simplificar los ejemplos conservaremos a la pantalla como metáfora principal de la interfaz de usuario.

2.1. Elementos básicos

Tomemos como ejemplo inicial el diseño de la pantalla de inscripción de un socio a un videoclub:



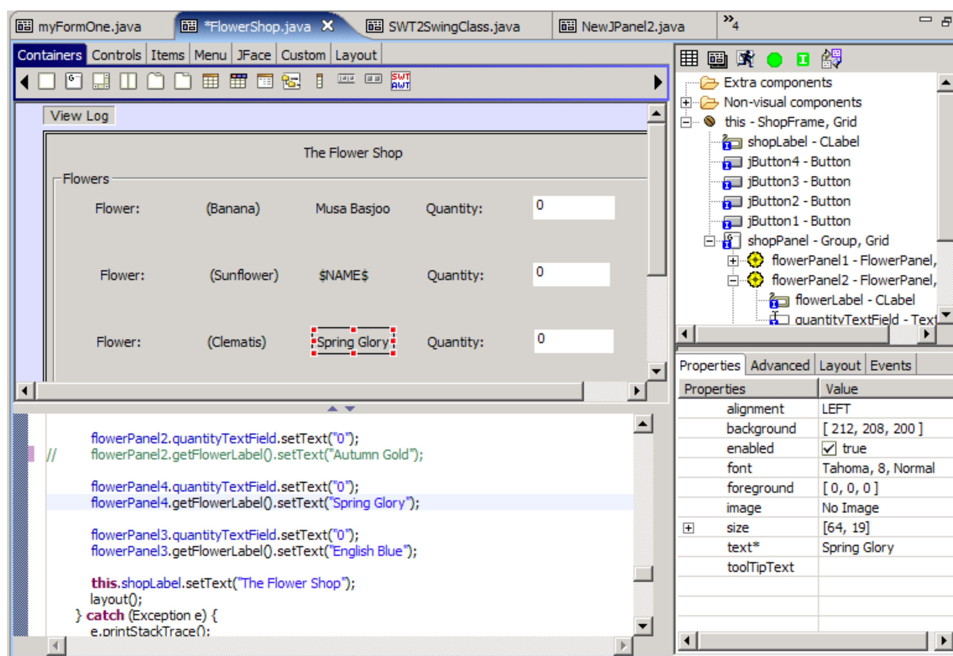
Independientemente de la tecnología que elijamos, para desarrollar la interfaz de usuario vamos a tener que trabajar sobre algunos puntos:

- elementos gráficos
- layout de la pantalla
- binding y transformación de los datos
- manejo de eventos
- inicio y navegación del caso de uso
- manejo del estado de una pantalla y pasaje de información entre pantallas

Para poder construir una interfaz de usuario, se nos hace muy cuesta arriba si no contamos con componentes básicos que son los que permiten diseñar una ventana: botones, campos de texto, combos, etc. A estos componentes los vamos a llamar **elementos gráficos, controles o *widgets***.

Dentro de la parte estrictamente gráfica, hay otras cuestiones a resolver: cómo acomodamos los elementos gráficos (qué va "arriba" y qué va "abajo", o qué va "centrado"), en qué color se muestra, con qué font, etc. Todas esas cuestiones entran dentro de lo que llamaremos **layout**.

En algunos casos contaremos con herramientas visuales para definir el layout, en otros lo haremos en forma programática y en el mejor de los casos tendremos ambas opciones donde nosotros podremos decidir cuándo usar una herramienta provista por la IDE y cuándo trabajarla a través del código.



Manipulación visual para definir el layout de un formulario

Aún con todas las cuestiones gráficas resueltas, no tengo un sistema que interactúa con usuarios.

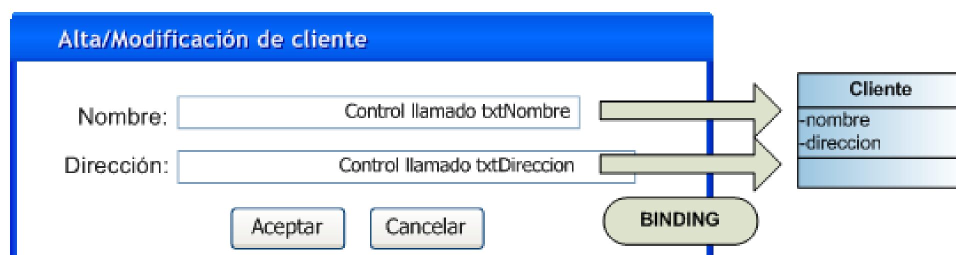
¿Qué falta? que los elementos gráficos estén relacionados con el dominio.

- podemos relacionar atributos de los objetos de dominio con controles.
- pero también podemos relacionar los eventos que se disparan con otro tipo de controles.

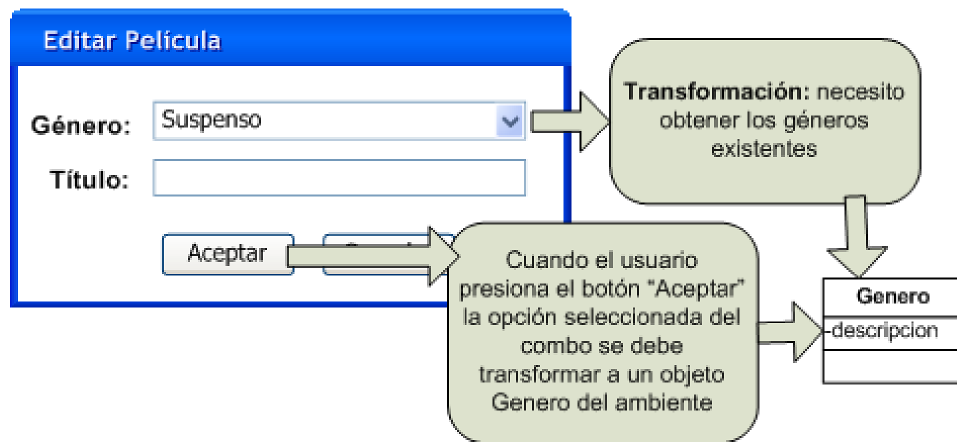
A esta relación la vamos a llamar **Binding**.

O sea muchos elementos gráficos son representaciones visuales de elementos de mi dominio, que pueden ser información (una película, la fecha de un alquiler) o acciones (el alta de un socio, la búsqueda de películas según un criterio, etc).

De alguna manera tenemos que vincular los datos que ingresa el usuario (en este caso puntual, el nombre y la dirección de un cliente) con objetos del dominio (un objeto Cliente con atributos nombre y dirección). Entonces un cuadro de texto (textbox) txtNombre contiene el valor del atributo nombre de un objeto cliente (cada control visual almacena un valor de algún atributo de un objeto de dominio).

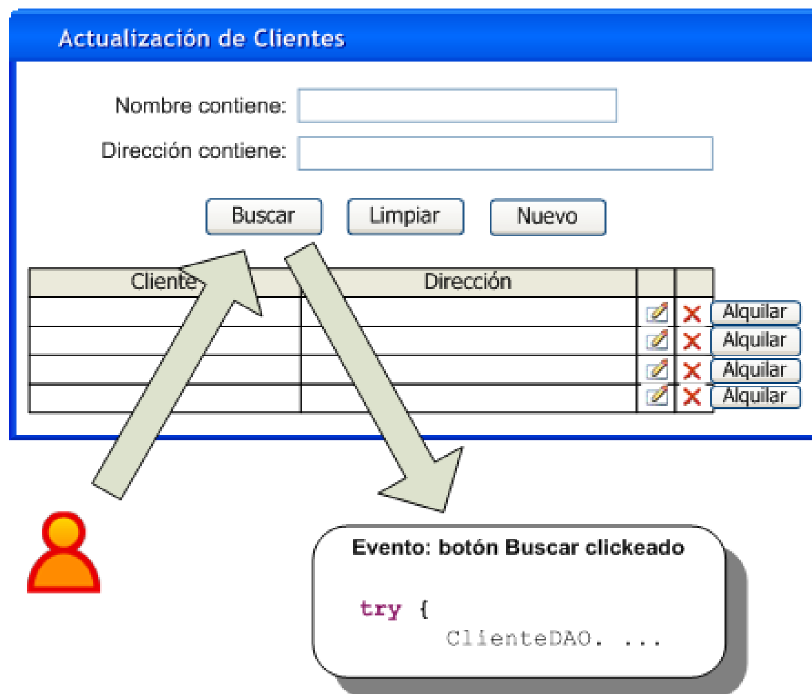


En algunos casos el binding no será tan directo entre objetos de dominio y controles y necesitaremos hacer una transformación para mostrar la información de los datos como el cliente los necesita:


Editar Película
Género:
Suspenso
Título:
Aceptar
Transformación: necesito obtener los géneros existentes
Cuando el usuario presiona el botón "Aceptar" la opción seleccionada del combo se debe transformar a un objeto Genero del ambiente
Genero
-descripcion

Binding de acciones o Manejo de eventos: se da por

- **eventos que se disparan automáticamente** por el ciclo de vida que tiene una pantalla: cuando se inicializa, cuando se activa el formulario o algún control, etc.
- **eventos que dispara el usuario**: al presionar el botón Aceptar, al elegir una opción en un combo, al hacer click en un radio button, al presionar Enter en un textbox, al hacer Drag & Drop sobre algún control, etc.

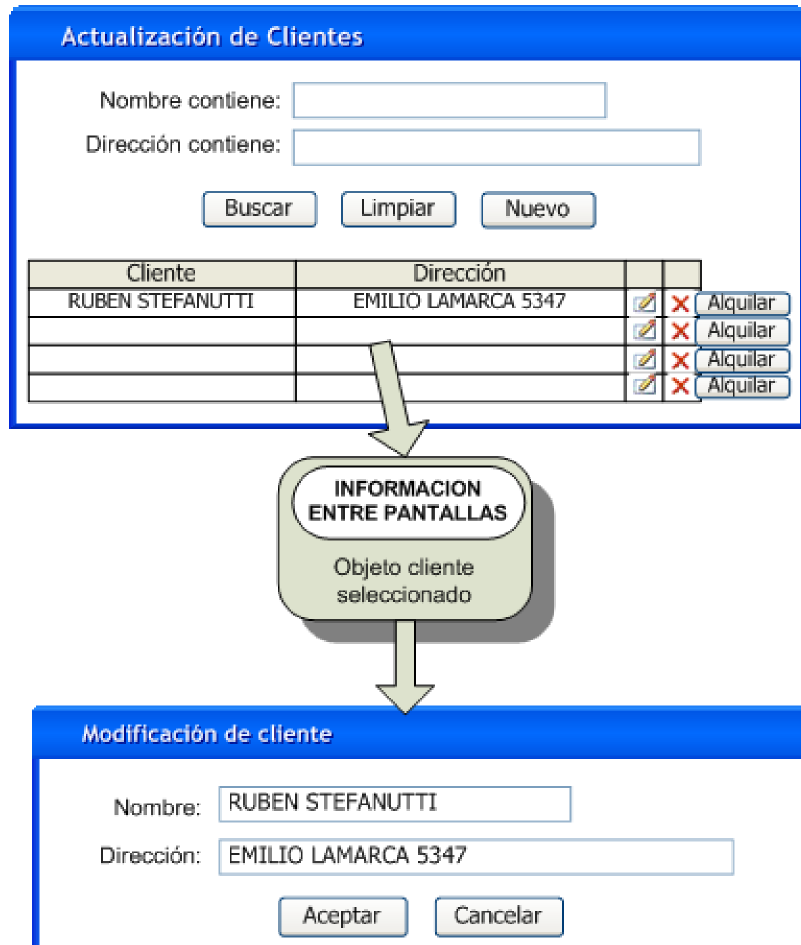


Inicio y Navegación del Caso de Uso: cada caso de uso tiene un esquema de navegación que estará representado por una o varias pantallas que se enlazan entre sí.

En el caso del alta de un nuevo cliente es una simple pantalla con la posterior confirmación al usuario de que los datos fueron correctamente ingresados. En el alquiler de películas tenemos un circuito en forma de asistente donde se enlaza la pantalla inicial (paso 1) con la siguiente (paso 2). Desde el paso 2 podemos avanzar a la pantalla final (paso 3) o bien retroceder al paso inicial.

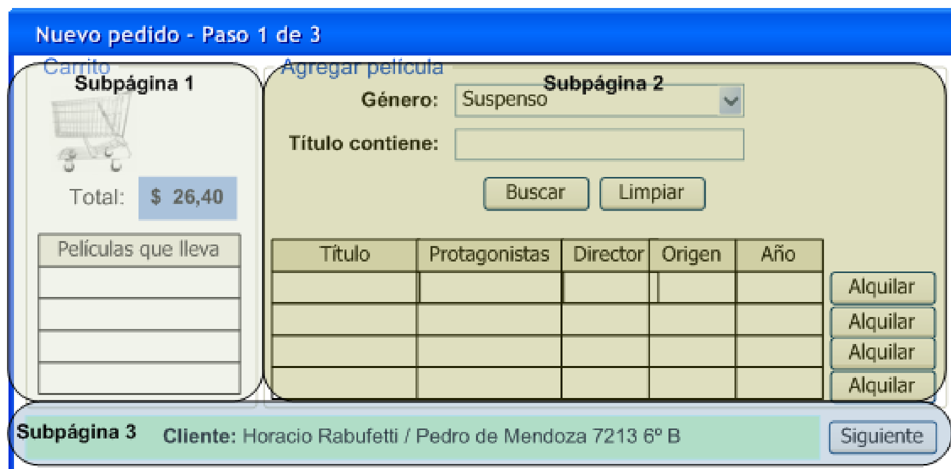
Estado de la pantalla e información pasada entre pantallas: cuando una pantalla depende de la selección de un dato en una pantalla anterior (o

de anteriores como en el caso del asistente), debemos considerar de qué manera podemos pasar o compartir información entre pantallas.



Una característica deseable en una tecnología de presentación es que se pueda dividir una pantalla en varias partes, contemplando la posibilidad de:

- compartir o enviar información entre las subpáginas que la componen
- que una página pueda responder ante eventos originados en otra página



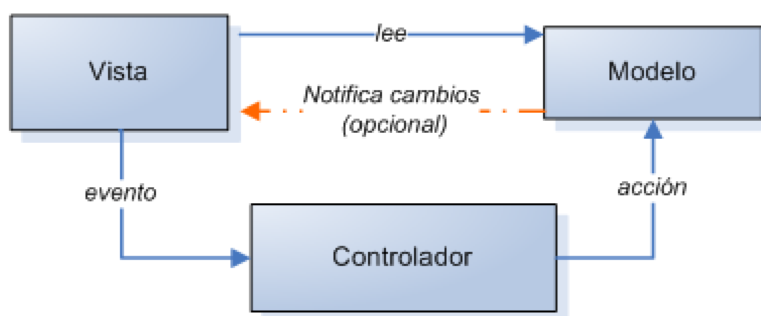
El paso 1 del asistente para generar un nuevo pedido se divide en 3 subpáginas: una donde muestra el carrito de películas que lleva, otra que permite buscar y alquilar una película y la tercera que muestra los datos del cliente que va a alquilar

2.2. MVC

Al introducir el concepto de binding hemos separado:

- por una parte el *modelo* (los que hemos llamado objetos de negocio en la introducción). Ejemplos de este tipo de abstracciones: el objeto que representa al cliente de un videoclub, una película, un alquiler, etc.
- y la interfaz visual que utilizará el usuario para ver/actualizar la información. Esta parte es llamada *vista* y dependerá de la tecnología de presentación que utilicemos.

Esta idea surge con el Smalltalk 80 [1] [3] [4] junto con un tercer actor: el controlador, que es el encargado de manejar los eventos del usuario y ejecutar acciones sobre el modelo.



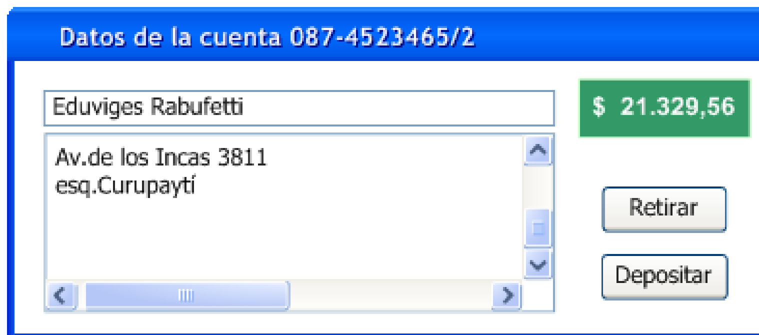
Un modelo puede tener asociados cero, uno o varios pares vista/controlador. Mencionamos algunas ventajas:

- al separar responsabilidades, es factible dividir el trabajo entre dos personas (teniendo en cuenta que la comunicación entre el modelo y el par vista/controlador tiene varias soluciones conocidas la interfaz entre ambos es mucho más fácil de definir).
- se pueden definir diferentes vistas para un mismo modelo sin tener que modificar los objetos de negocio (de otra manera hay que desarrollar dos vistas similares copiando y pegando, con las consecuencias que esto implica)

El modelo puede disparar notificaciones de cambios a sus vistas, lo cual le da al esquema MVC una ventaja adicional respecto al desarrollo de interfaces de usuario donde no hay división de responsabilidades: la vista se mantiene siempre actualizada (sin necesidad de obligar al usuario a refrescar

la pantalla). Lo veremos a continuación con un ejemplo: tenemos una aplicación bancaria que permite hacer un seguimiento de las cuentas que tienen los clientes.

Hay dos usuarios: el cajero que puede consultar y realizar acciones sobre los clientes del banco y un director de cuentas que monitorea los clientes que tiene a su cargo.



Pantalla EditarClienteView

Seguimiento de clientes / Jefe de cuentas Alberto Paruzzo

Cliente	Dirección	Saldo	
Eduviges Rabufetti	Av.de los Incas 3811	21.329,56	...
Cosme Fulanito	Ruta 7 km.328	17.506,44	...
			...
			...
			...

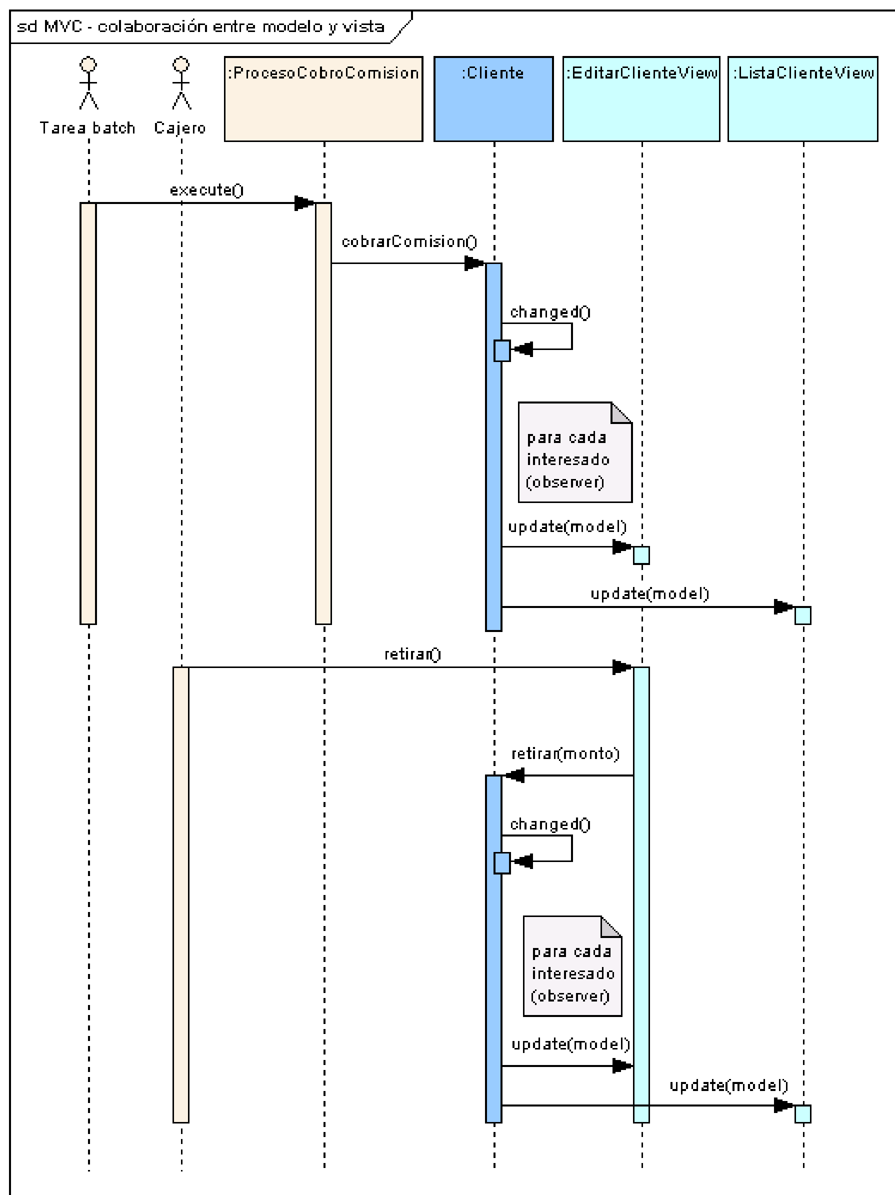
Pantalla ListarClientesView

Se plantean dos escenarios:

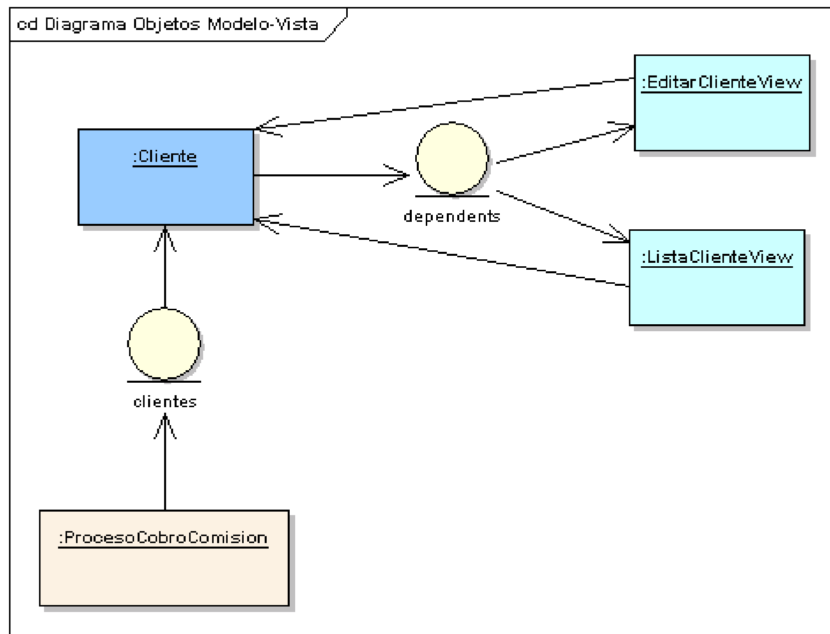
1. El último día del mes el banco le cobra el costo de mantenimiento de la cuenta (a través de un proceso que se ejecuta automáticamente)

2. El cliente va al banco y retira $ 30 de su cuenta

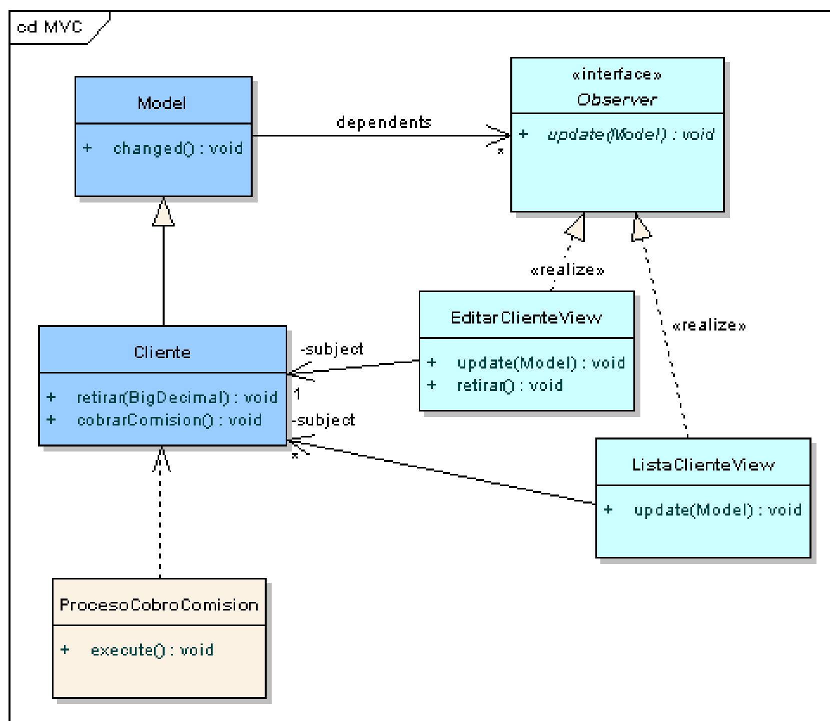
En el primero de los casos, la actualización se produce por el proceso mensual que calcula la comisión a cobrar a los clientes. En el segundo de los casos, es la vista que usa el cajero del banco la que modifica al objeto cliente, enviándole el mensaje `retirar()`. En ambos casos el cliente dispara un evento `changed()` a las vistas interesadas:



Vemos en un diagrama de objetos cómo colaboran los objetos de dominio y las vistas:



Lo interesante es que para notificar cambios el modelo no conoce a las clases concretas que implementan las vistas, sino que habla con una interfaz (el Observer):



Las vistas *sí* conocen al cliente, trabajan con las instancias concretas de los objetos de dominio (sería raro que quisiéramos desacoplar totalmente a la vista del dominio, pensemos que si estamos haciendo la pantalla de consulta de un cliente, es esperable que la vista esté acoplada al objeto cliente).

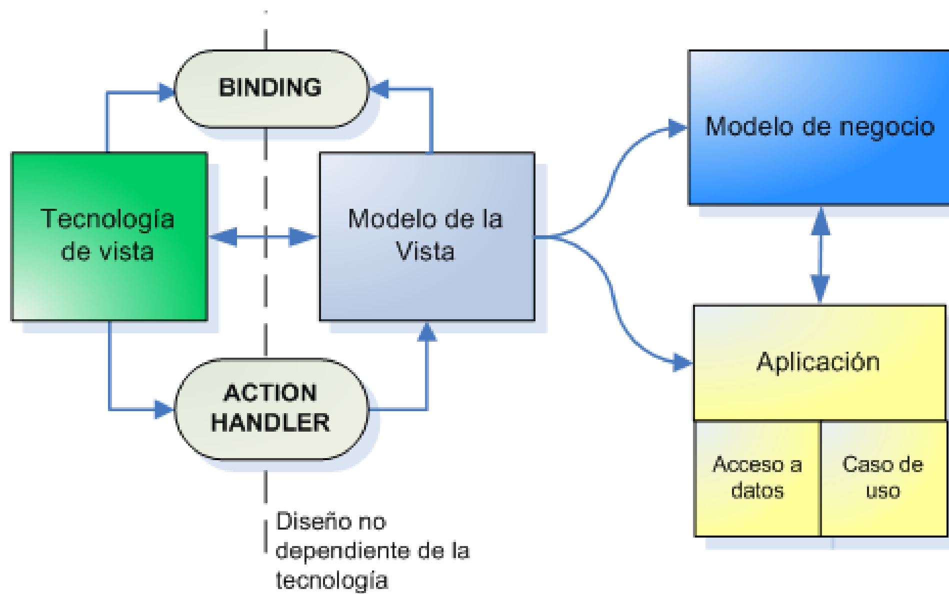
2.2.1. Extendiendo el MVC

Desde entonces han aparecido muchas variantes al esquema original[1]. Lo que es interesante señalar del MVC es que separar pantalla y modelo es el primer paso para encontrar otras abstracciones:

- la respuesta ante los eventos (action handlers),
- el binding que enlaza la vista con el dominio y viceversa,
- la lógica para el armado de la vista (que llamamos modelo de la vista): navegación, manejo de información entre pantallas y layout, que puede incluir el desarrollo de controles visuales propios o el uso de controles existentes.
- la vista tendrá que interactuar tanto con los objetos de dominio (el modelo) como con otros objetos que proveen determinados servicios (presentados en la introducción del apunte como objetos de aplicación): los objetos de acceso a los datos y los que representan un caso de uso. Los primeros nos permiten filtrar los clientes que vivan en la calle Nazca o encontrar qué clientes vieron la quinta temporada de Lost. Los segundos pueden determinar cómo se ingresa un cliente en el videoclub o cómo se actualizan los datos de facturación de un cliente (donde el cómo incluye cosas que no forman parte del negocio).

El sentido práctico es que además de no estar acoplados a la tecnología, también podamos llevar esos *concerns* a un lenguaje que sea extensible y conocido por nosotros, por ejemplo en uno que sea orientado a objetos.

[1] Puede estudiarse el capítulo dedicado al Observer en [6] donde se comentan variantes al esquema de notificaciones original entre modelo y vista para mejorar la eficiencia



Entonces, si tengo estos requerimientos:

- los clientes nuevos no pueden llevar estrenos
- mostrar los estrenos en color azul
- en todas las pantallas donde se muestre una película hay que visualizar el título, el año (y quizás también quiera agregar datos sobre el director)
- ordenar los estrenos primero en lugar de alfabéticamente

la ventaja que tenemos al dividir el modelo de la vista de la tecnología en la que se construye esa vista es que es mucho más factible encontrar un único lugar donde codificar esa responsabilidad (relacionado con la cualidad de diseño *once and only once*). Llevarlo al terreno de objetos presupone aprovechar las ventajas de este paradigma (principalmente en cuanto al polimorfismo y la herencia).

2.3. Pasando en limpio

Queremos que no haya cuestiones de presentación en el negocio:

- Porque suele ser la parte más compleja
- Porque es difícil encontrar buenas abstracciones, porque es la parte más inmadura de la ciencia

- Porque es donde más restricciones de usuario podemos tener
- Porque es donde más restricciones tecnológicas podemos tener

Queremos que no haya cuestiones de negocio en la presentación:

- Porque se dispersa el conocimiento y luego para cambiar algo tengo que tocar múltiples lugares distribuidos entres las capas y está la lógica de negocio (probablemente simple) diluida en cuestiones más complicadas.
- Porque pierdo cohesión del dominio, luego se complican los esfuerzos para testeo unitario del negocio: o bien se pierde la unitariedad pasando a tests de aceptación; o bien pierdo el once and only once entre tests y UI.

Abstraer es distinto que no se conozcan: Una idea bastante instalada en el mercado es abstraer la presentación del dominio tanto como sea posible. De hecho algunos piensan que lo mejor es que no se conozcan/ni se hablen. Nuestra idea es que la presentación no sólo hable con el dominio sino que le pida todo lo que le tenga que pedir.

Para profundizar sobre lo dicho anteriormente pueden verse los apuntes de la cátedra sobre las tecnologías SWT y JSP/Servlets.

3. Aplicación

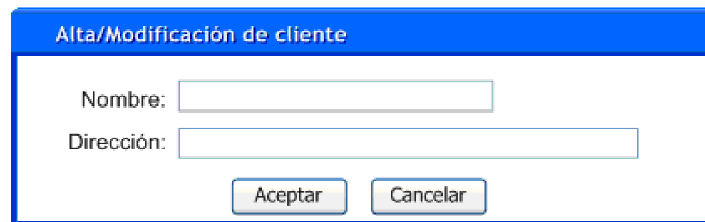
Si la interfaz de usuario es el punto de entrada hacia nuestro sistema, el concern de aplicación (que ocupará el presente capítulo) es el que permite relacionar la parte visual que el usuario manipula con el resto de las cosas que debe hacer el sistema.

3.1. Definición

El código de aplicación sirve como enlace entre los objetos de negocio, los que modelan el caso de uso y los que trabajan con temas específicos de la tecnología. Ellos permiten contestar la pregunta *qué sucede cuando se aprieta el botón Aceptar*, porque justamente relacionan cada uno de los objetos que son necesarios para llevar a cabo una funcionalidad del sistema.

3.2. Un ejemplo

Modelamos el ingreso de un nuevo cliente al videoclub. La presentación es una pantalla donde se ingresa nombre y dirección:



El requerimiento pide que los campos nombre y dirección no queden en blanco para poder ingresar un nuevo socio. Veamos qué responsabilidades hay que resolver:

- validar que el nombre y la dirección estén cargados
- instanciar un nuevo cliente
- agregar el nuevo cliente al repositorio

Para instanciar un nuevo cliente, la clase Cliente puede ofrecer un constructor adecuado

```
public Cliente(String nombre, String direccion) {
   ...
```

Pero ¿quién dispara el alta de un socio? O sea, dónde ubicamos la responsabilidad de validar, instanciar al cliente y luego guardarlo en el repositorio. Algunas opciones:

1. asignar toda la responsabilidad al botón Aceptar. Si bien es una solución simple, estamos mezclando tareas de interfaz de usuario y de aplicación. Esto dificulta pensar en el ingreso de un socio desde otro contexto (por ejemplo si desarrollamos una carga masiva de socios importándolos a través de un archivo, o si necesitamos armar otra pantalla en la misma tecnología desde la cual asociar un nuevo cliente). Además, como dijimos antes, la interfaz de usuario pierde cohesión: hace muchas cosas, no sólo presenta la información para que el usuario la manipule sino que también resuelve validaciones, se conecta con el repositorio e instancia los objetos de negocio. De esa manera los tests ya no son tan unitarios (porque el método Aceptar está haciendo muchas cosas a la vez) y cuando hay un error se vuelve más complicado encontrar el origen del problema.

2. pensamos en un objeto que resuelve el caso de uso como un 'servicio' (donde el servicio no tiene estado). Si bien puede estar implementado como un objeto, se parece mucho más a una función o a un procedimiento de los paradigmas imperativos. La ventaja respecto a la opción anterior es que puedo invocar cada servicio desde diferentes contextos. La desventaja es que como el servicio no funciona como objeto, no puedo mantener una conversación: tengo que acopiar toda la información para enviarla tal cual el servicio la necesita.

3. abstraemos el caso de uso o requerimiento como un objeto. Aquí tengo la posibilidad de que la interfaz de usuario le envíe varios mensajes al caso de uso manteniendo una conversación, el objeto caso de uso tiene estado interno y en muchos casos ahorramos el trabajo a la pantalla de armar toda una estructura que el servicio necesita para resolver el requerimiento.

3.2.1. Objetos caso de uso: ingreso de un socio

En nuestro caso vamos a elegir modelar un objeto que representa al ingreso de un nuevo socio (le podemos poner nombre: IngresoSocio). Este mismo objeto es el que guía el proceso de creación de un nuevo cliente.

```
>>IngresoSocio
public void ejecutar(...parametros...) {
```

```
    this.validar(...parametros...);
    Cliente cliente = new Cliente(...nombre..., ...direccion...);
    agregar el cliente instanciado al repositorio
}
```

Algunas opciones sobre los parámetros que recibe el método ejecutar:

- `public void ejecutar(String nombre, String direccion) {`

  Nombre y dirección por separado. La desventaja principal que tiene este approach es que el objeto cliente se desmembra en una estructura lineal fácilmente permeable a cualquier cambio (no sólo si necesitamos agregar el CUIT al cliente, también si la dirección deja de ser un string para convertirse en un objeto Direccion)

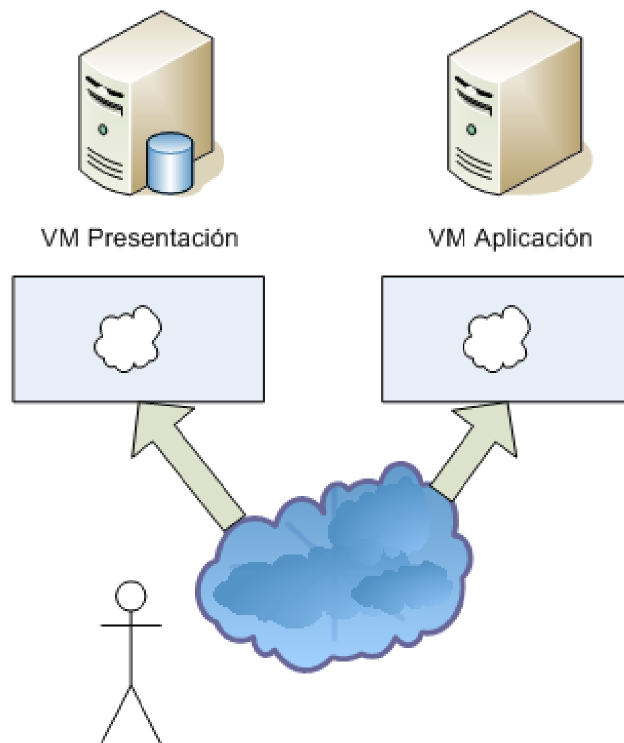
- `public void ejecutar(Cliente cliente) {`

  Un objeto cliente es algo bastante más saludable, no sólo porque los cambios no afectan a la interfaz del método ejecutar que tiene el caso de uso sino porque la presentación crea al objeto cliente cuando hace el binding, entonces 1) llamar al constructor en el caso de uso ya no es necesario, 2) el objeto cliente que genera la presentación es el mismo objeto que el que recibe el caso de uso, por lo tanto ambos pueden aprovechar el comportamiento que define el objeto de negocio (recalquemos la importancia de saber que **son el mismo objeto**).
  Si trabajamos en una arquitectura donde presentación y aplicación viven en el mismo ambiente ésta debería ser la alternativa más natural.

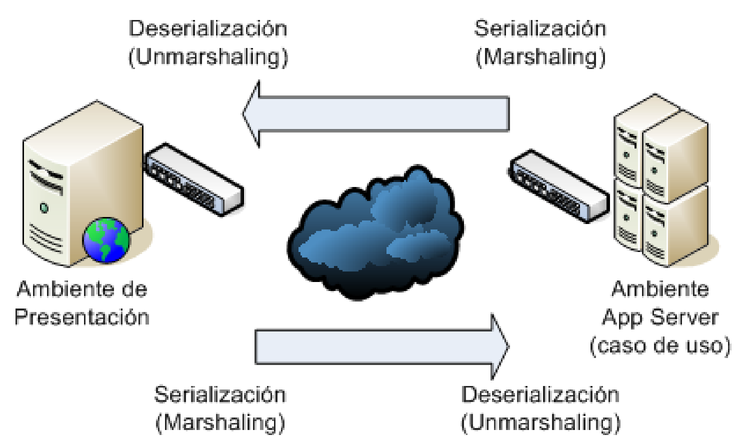
- `public void ejecutar() {`

  Si el objeto caso de uso tiene estado interno, podríamos separar dos momentos: 1) el envío de parámetros (mediante mensajes setters: setCliente) y 2) la ejecución del caso de uso. Así la interfaz del método ejecutar() ya no requiere parámetros. Si por el contrario elegimos modelar el caso de uso como un objeto sin estado (stateless), no contaremos con esta posibilidad. Más adelante volveremos sobre esta idea.

- cuando la presentación y el caso de uso viven en ambientes distintos (porque la naturaleza del problema es distribuido) recibir un objeto cliente no es trivial. En ambientes distribuidos se torna complicado respetar la identidad de un objeto, porque hay que representar el mismo concepto en dos VM diferentes.



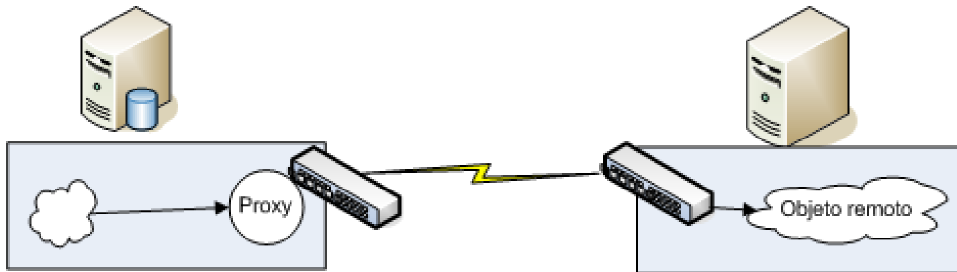
Esto significa que desde la VM donde reside el caso de uso hay que generar un objeto cliente que luego debe serializarse, enviarse y deserializarse en la VM de presentación (y viceversa cuando la presentación actualiza la información del objeto y quiere sincronizarla con el otro ambiente).



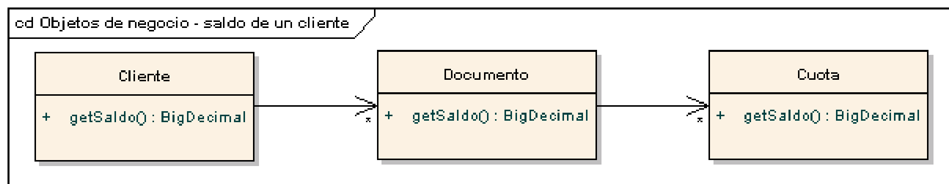
3.2.2. Objetos Proxys

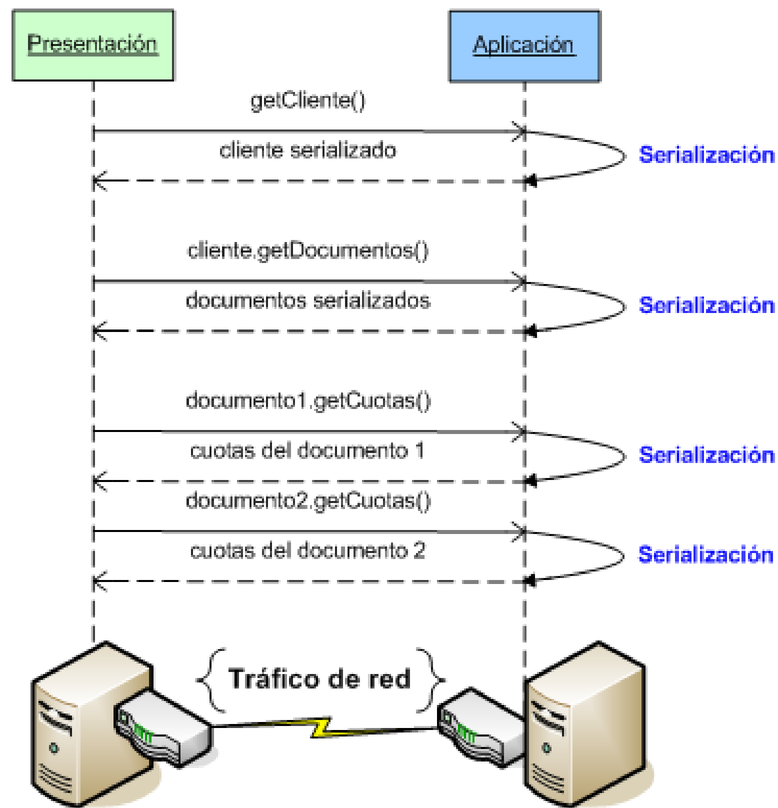
Implementar la serialización de un objeto no trae mayores dificultades (en la mayoría de los lenguajes ya hay librerías que se encargan de esta tarea). No obstante, un objeto cliente puede tener asociado reservas y alquileres de películas, una cuenta corriente (que incluye pagos y documentos de deuda), calificaciones de las películas que vio, etc.

Entonces cuando estoy pasando un cliente, ¿cuál es el límite de la serialización? ¿Debemos incluir los objetos relacionados, alguno de ellos o sólo el objeto cliente?



Una estrategia al trabajar con objetos remotos es serializar en forma *lazy*, es decir, utilizar un objeto que sirva como intermediario (o proxy) que pida los objetos remotos a medida que los necesita. El inconveniente que trae esto es que cuando necesitamos traer mucha información el tráfico de red se incrementa y en consecuencia la performance se degrada.

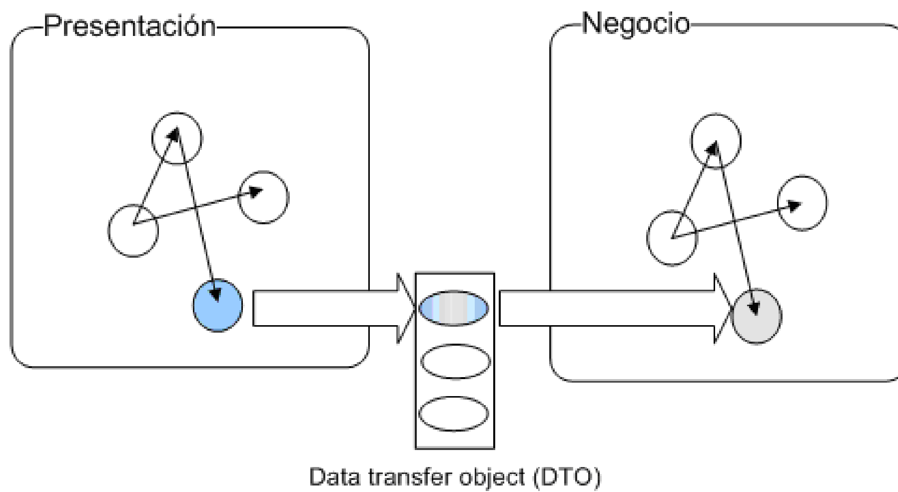




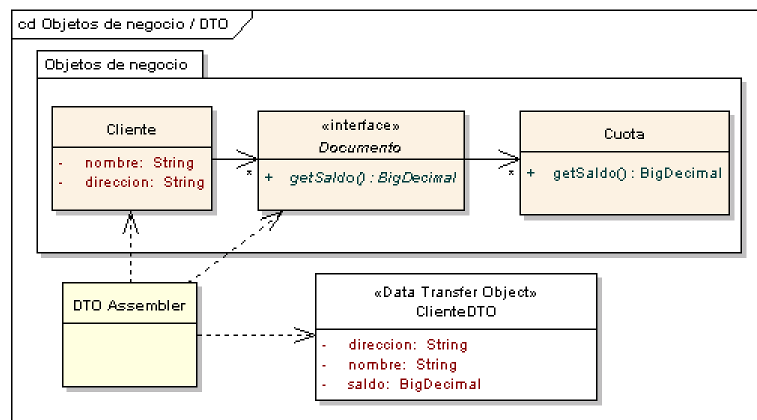
3.2.3. Objetos DTO

Entonces quizás me sirva modelar la transferencia de la información como un objeto: si el cliente tiene nombre, dirección y el saldo que se obtiene sumarizando los montos de cada documento (positivos o negativos), en lugar de serializar al cliente y a su colección de documentos (que a su vez tiene referencias a un conjunto de cuotas) se crea un objeto intermedio para transferir la información como la necesita algún caso de uso: un **ClienteTransferObject** o **ClienteValueObject**[2]

[2]Algunos estudiosos de la materia como Martin Fowler usan el término Value Object para objetos que modelan conceptos simples -similares a los tipos primitivos de Java: un rango de fechas o un monto de dinero- cuya igualdad no está basada en la identidad (http://martinfowler.com/eaaCatalog/valueObject.html). Por este motivo para evitar confusiones sólo utilizaremos el término Data Transfer Object para referirnos a los objetos que modelan la transferencia de información.



La identidad se conserva mediante algún atributo unívoco para todos los ambientes (supongamos en este ejemplo, un id de tipo Integer).



El objeto intermedio no tiene comportamiento, sólo guarda alguno de los atributos del objeto original a los que se accede mediante getters y setters. Muchas veces suelen definirse a los DTO como objetos inmutables, codificando sólo un constructor que acepte todos los parámetros que las propiedades necesitan y los getters correspondientes (sin métodos setter).

Volviendo sobre el método ejecutar, el caso de uso quedaría definido así:

```
public void ejecutar(ClienteDTO clienteDTO) {
```

```
        this.validar(clienteDTO);
        Cliente cliente =
            new Cliente(clienteDTO.getNombre(),
                        clienteDTO.getDireccion());
        agregar el cliente instanciado al repositorio
}
```

Ventajas: El objeto de transferencia reduce el envío de mensajes entre ambos ambientes, los datos viajan por la red una sola vez. Al encapsular la transferencia también podemos agregar o sacar información sin modificar la interfaz del método ejecutar().

A tener en cuenta: Debe considerarse que este tipo de objetos introducen redundancia en el modelo: el objeto Cliente y ClienteDTO se parecen mucho, tanto que el segundo es un cliente 'disecado', sin lógica, solamente una estructura de datos aplanada. Si nuestra idea es que la presentación reciba un ClienteDTO para mostrar información en la pantalla, tenemos que pagar el costo de no poder enviarle mensajes al objeto de negocio. Algunas cosas pueden ser triviales: existe un método esMoroso() que indica si el cliente debe plata. Podemos definir un método esMoroso() en ClienteDTO: de esa manera estamos duplicando la lógica de negocio, lo sabemos, pero es apenas un método. Ahora qué pasa si tenemos un requerimiento como el siguiente: 'un cliente puede reservar estrenos si no tiene facturas de más de 3 meses impagas...'

La desventaja es que para comunicar dos ambientes OO estamos utilizando una técnica que descarta la principal abstracción del paradigma (objeto agrupando atributos y comportamiento).

Debemos entonces justificar el costo/beneficio de utilizar DTOs en cada uno de los escenarios posibles. El contexto puede variar dependiendo de si estoy haciendo una aplicación donde hay un solo ambiente, donde hay varios o bien cuando estoy ofreciendo un servicio y no tengo idea si los consumidores de ese servicio están trabajando en tecnología de objetos.

Seguimos entonces con el caso de uso...

Para validar que los campos no sean vacíos, podemos tener un método que haga ese control:

```
public void validar(String nombre, String direccion) {
```

```
  if (nombre.equals("") {
     throw new ValidationException("Debe ingresar nombre");
  }

  if ...
}
```

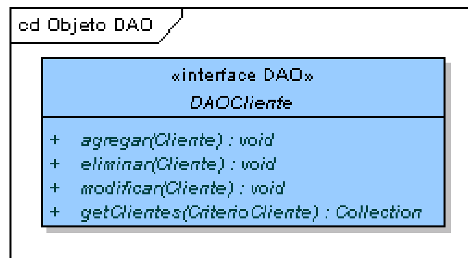
Es responsabilidad de la interfaz de usuario capturar este error y mostrar un mensaje representativo: en este caso se trata de un error de negocio, lo mejor es atrapar en un catch el error de validación y delegar en el error el mensaje a mostrar:

```
>>Interfaz de usuario, método que se ejecuta al
>>apretar el botón aceptar (depende del framework)
public void ejecutar() {
   try {
      ...
   } catch (ValidationException e) {
      this.mostrarError(e.getMessage());
      // muestra un Message Box en pantalla (o similar)
   }
}
```

3.2.4. Objetos DAO

Para ingresar un nuevo socio al conjunto de socios del videoclub entra en juego un objeto que maneja la responsabilidad de acceder al repositorio, independientemente del formato en el que se almacenen esos datos. Por ese motivo recibe el nombre **Data Access Object**. En definitiva necesitamos las funciones básicas de cualquier estructura de datos:

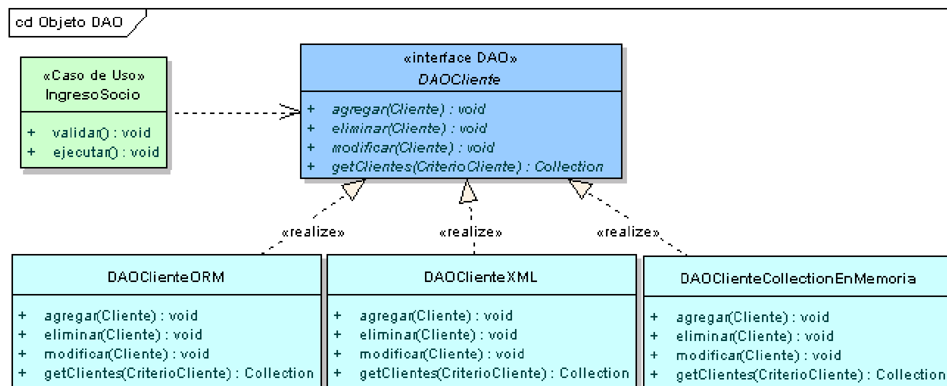
- agregar
- eliminar
- actualizar
- recuperar un elemento, en base a un criterio de filtrado
- y opcionalmente querer recuperar un elemento solo



La interfaz del DAO nos sirve para encapsular la forma en que accedemos al repositorio:

- el repositorio podría ser una base de objetos
- o bien un archivo XML
- o bien una base de datos relacional...
  - ...y accederse mediante JDBC
  - o bien usando un mapeador O/R
  - o cualquiera de las tecnologías que veremos en el apunte de Persistencia

nosotros definiremos una interfaz para que el objeto IngresoSocio use al DAOCliente.



Algunas consideraciones de implementación de los DAOs:

- tanto al agregar, como al modificar o eliminar, recibimos un parámetro Cliente. Esto dependerá -como hemos visto anteriormente- de nuestra decisión de trabajar con objetos de transferencia o de negocio.
- En algunas implementaciones [7], se sugiere que el método agregar() devuelva el Id del objeto nuevo

```
public interface CustomerDAO {
   public int insertCustomer(...);
```

y que el eliminar devuelva un boolean que indica si pudo o no eliminar al objeto en cuestión,

```
public boolean deleteCustomer(...);
```

La primera decisión es cuanto menos discutible: hacer que el alta de un cliente devuelva un identificador entero está revelando cierta dependencia con la tecnología elegida. Esto puede ser razonable en una base de datos relacional donde cada registro requiere un identificador unívoco y se puede definir un número autoincremental, pero es irrelevante en bases de objetos o cuando elegimos guardar un snapshot del ambiente en un archivo. En todo caso si estamos pasando un objeto cliente y el pasaje de parámetros es por referencia, el DAO puede setear el identificador en un atributo del objeto Cliente (setId()).

Por otra parte en caso de no poder eliminar un cliente o modificarlo, lo que estamos esperando es saber exactamente cuál fue el error (sea éste de negocio o de programa), así que no necesitamos un boolean (que provee escasa información sobre la característica del error) sino esperar que haya una excepción que nos cuente qué pasó.

El DAO actúa como un *strategy* que desacopla el caso de uso y la forma en que se persiste el objeto: podemos intercambiar los DAOs sin cambiar la esencia del caso de uso. Hemos visto que no todas las decisiones necesitan ser dinámicas, así que posiblemente estemos sobrediseñando un poco nuestra solución: abstraemos una interfaz general para tener quizás sólo una implementación (además de la indirección que supone para el caso de uso). De todas maneras recalquemos que como ingenieros nos pagan por tomar decisiones: entonces podemos asumir el costo de dejar esa puerta abierta (siempre que seamos capaces de entender que el beneficio lo vale).

3.3. Otras responsabilidades del caso de uso

- **Transaccionalidad**: si un caso de uso involucra varias operaciones que deben ejecutarse en conjunto, es responsabilidad del objeto caso de uso asegurar que todas las operaciones se completen (o en caso contrario ninguna de ellas). El DAO no puede asegurar que la transacción

se complete porque trabaja a nivel de operación (una inserción, una modificación, una eliminación, etc)

```
public void ejecutar(Cliente cliente) {
     this.beginTransaction();
     try {
        this.validar(cliente);
        dao.agregar(cliente);  // el dao ya está instanciado
        this.commitTransaction();
     } catch (...Exception e) {
        this.rollbackTransaction();
        throw new ExceptionDeAltoNivel
           ("No se pudo asociar el cliente", e);
     }
  }
```

Las implementaciones posibles de los métodos beginTransaction(), commitTransaction() y rollbackTransaction() pueden variar, inclusive en lugar de demarcar el origen y fin de la transacción en forma programática podríamos trabajar con transacciones declarativas (delimitados por el comienzo y fin de cada método, en caso de error automáticamente haría el rollbackTransaction()). Lo importante es asegurar que todas las operaciones del caso de uso ocurran a la vez o bien ninguna (para que la aplicación no quede en un estado inconsistente). Relacionado con este concern está el **acceso concurrente de los usuarios**. Para más información puede verse [8] y [9].

De la misma manera, aparecen otras cuestiones como:

- **Seguridad**: determinar si el usuario logueado puede ejecutar un caso de uso en base a un esquema de perfiles, contemplar mecanismos de encriptación/desencriptación de los datos sensibles que llegan o salen del ambiente (especialmente cuando tenemos ambientes distribuidos)
- **Auditoría**: obtener información contextual sobre la ejecución de un caso de uso para poder hacer cambios en caliente sobre el sistema
- **Distribución**: tener más de una capa física

En definitiva, lo que respecta a la **arquitectura** del sistema son temas que suelen incorporarse a las responsabilidades del caso de uso.

Ahora bien, nuestro desafío es que cuando tengamos que hacer los siguientes casos de uso:

- Ingreso de un cliente
- Modificación de los datos de un cliente
- Alquiler de una película

No tengamos que duplicar la lógica de lo que es un caso de uso en sí: todas las cuestiones de seguridad, auditoría, transaccionalidad, etc. no deberían estar duplicadas cada vez que construyo un objeto que representa un caso de uso.

Está claro que la implementación de cada caso de uso es lo que varía, ahora bien: ¿qué herramientas tengo para no duplicar código?

3.4. Extensiones al Caso de Uso

1. *Template method* Podemos abstraer una clase CasoUso de la cual hereden IngresoSocio, ModificacionSocio y AlquilerPelicula y trabajar con un template method: el caso de uso podría tener una definición como la siguiente:

```
public void ejecutar(...parámetros del caso de uso...) {
    try {
        this.inicializar();
        this.doEjecutar(...parámetros del caso de uso...);
        this.finalizar();
    } catch (Exception e) {
        this.manejarError();
    }
}
```

    - En el inicializar podemos comenzar la transacción, o bien loguear el comienzo, o agregar validaciones de perfiles de seguridad en base al usuario, etc.
    - El doEjecutar() es un *método abstracto*, lo implementa cada caso de uso concreto.
    - El método finalizar cierra la transacción, o bien loguea el tiempo transcurrido para la operación, o bien audita que la operación se ejecutó, etc.

- El método manejarError podría rollbackear la transacción, o bien loguear que la operación no se ejecutó correctamente, o levantar una alarma, etc.

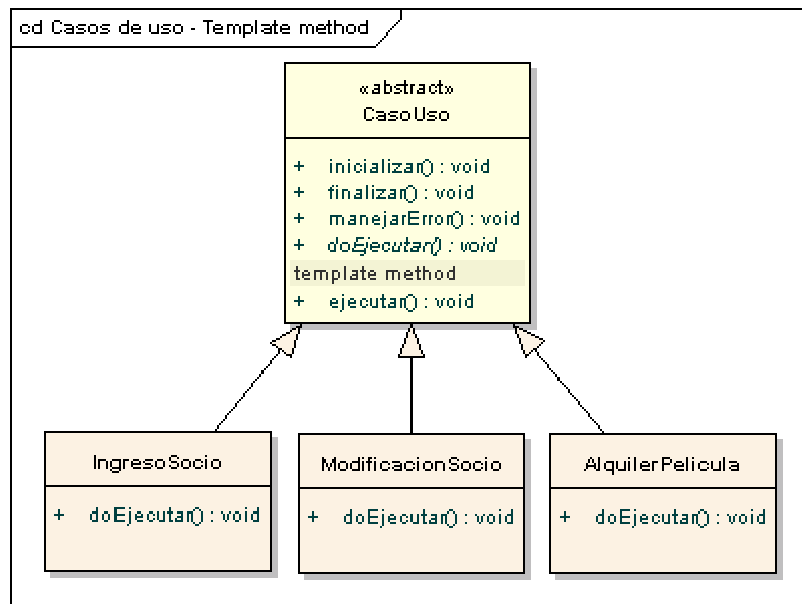
Todo esto está muy bien, pero estamos olvidando un detalle importante: el ingreso de un socio necesita los datos de un socio, el alquiler de una película necesita como input el socio y la película. En particular cada caso de uso necesita distintos parámetros, con lo cual el doEjecutar con diferentes parámetros pierde el polimorfismo.

Para conservar la interfaz común para todos los casos de uso, tenemos dos alternativas:
1) representar los parámetros del caso de uso mediante un objeto.

```
public void ejecutar(Map parametros) {
    try {
        this.inicializar();
        this.doEjecutar(parametros);
        this.finalizar();
    } catch (Exception e) {
        this.manejarError();
    }
}
```

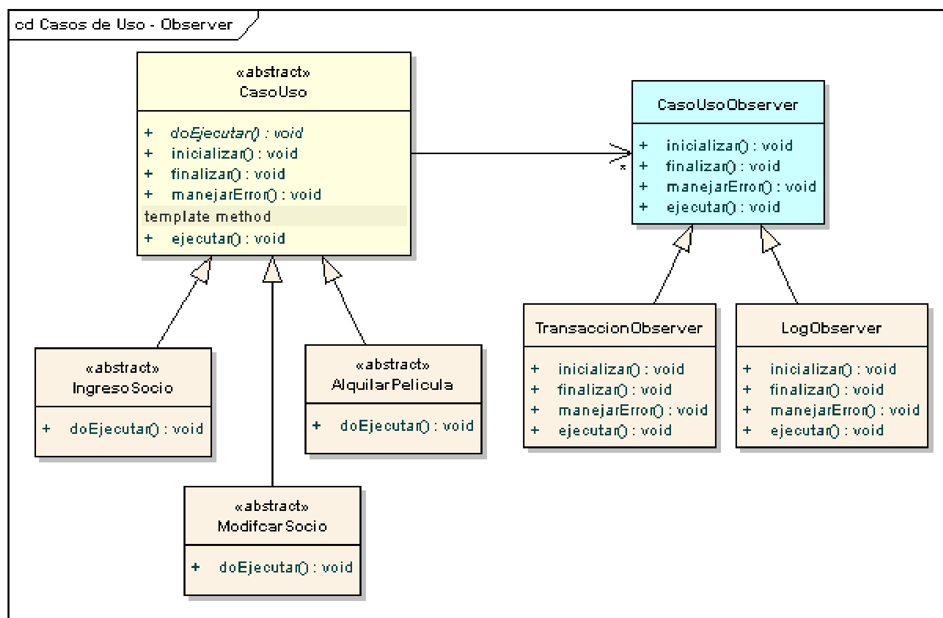
La desventaja de esta opción es que necesitamos agregar una abstracción más...

2) separar los momentos de envío de parámetros y de ejecución, de manera que el doEjecutar no necesite parámetros. Para lo cual retomamos la idea de abstraer los casos de uso como objetos stateful (con comportamiento **y** estado)



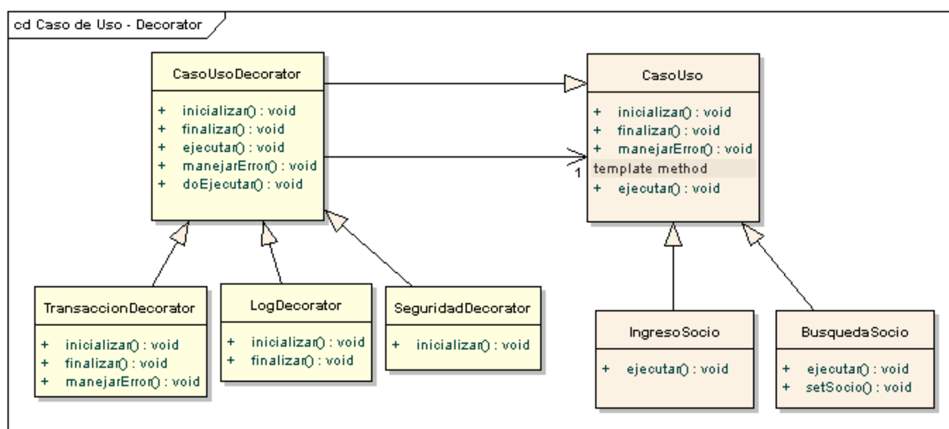
La ventaja de trabajar con el template method ejecutar es que el caso de uso define el comportamiento default de la arquitectura, mientras que cada caso de uso se concentra en la operación que tiene que hacer el sistema o bien puede redefinir el comportamiento de dicha operación cuando comience, termine o falle.

2. *Callbacks: Observers* El template method introducido permite fijar tres puntos para cada caso de uso del sistema: estamos insertando código en los momentos que hemos determinado (inicializar(), ejecutar(), finalizar() y manejarError()). Si queremos activar o desactivar diferentes comportamientos en tiempo de ejecución, podríamos trabajar con diferentes observers:



De esta manera no estamos mezclando en la inicialización las responsabilidades de transacción, logueo, seguridad, etc. La limitación que tenemos es que la cantidad de eventos que registramos está fija en el diseño de nuestra solución: sólo podemos agregar comportamiento en esos puntos.

3. *Interceptar código: Decorators* Otra alternativa es decorar el código de los casos de uso, separando de la misma manera las responsabilidades de cada concern en una clase concreta:



Los decorators ofrecen la flexibilidad de interceptar comportamiento antes o después de cada uno de los métodos definidos en la interfaz.

3.5. Límites

Los límites de este concern son difusos y no hay una respuesta homogénea para saber dónde comienzan y dónde terminan sus responsabilidades. Podemos decir que

- para algunos el código de la aplicación no debería estar atado a ninguna tecnología, estos objetos sólo representan los casos de uso como una capa de servicios. Entonces la transaccionalidad de una operación, e incluso la persistencia de la información modificada en dicho caso de uso es responsabilidad de otra capa.
- para otros el código de la aplicación se vale de elementos tecnológicos y de diseño para resolver un caso de uso.

La realidad es que es muy común que algunos requerimientos se mantengan en una zona gris:

No nos quita el sueño identificar si es 'de Presentación' o 'de Negocio' el código que hay que tocar, en todo caso lo que tenemos que rescatar es que el objetivo final es **no duplicar la lógica**.

Con cualquiera de las definiciones que tomemos la conclusión es similar: hay responsabilidades que cubrir. Nuestro objetivo como diseñadores es que haya un objeto (y sólo uno) que haga ese trabajo... una sola vez.

Referencias

[1] STEVE BURBECK: *Applications Programming in Smalltalk-80: How to use Model-View-Controller*, Paper, 1987-92.

[2] CHAMOND LIU: *Smalltalk, Objects and Design*, Editorial toExcel, 1996.

[3] TRYGVE REENSKAUG: *Thing-Model-View-Editor. An example from a planning system*, Paper, 1979

[4] TRYGVE REENSKAUG: *The Model-View-Controller (MVC). Its Past and Present*, Paper, 2003

[5] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON Y JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[6] SHERMAN R.ALPERT, KYLE BROWN Y BOBBY WOOLF: *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998

[7] SUN DEVELOPER NETWORK: *Core J2EE Patterns - Data Access Object*, Web oficial de Sun

[8] SUN DEVELOPER NETWORK : TRANSACTIONS: Web oficial de Sun http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Transaction.html

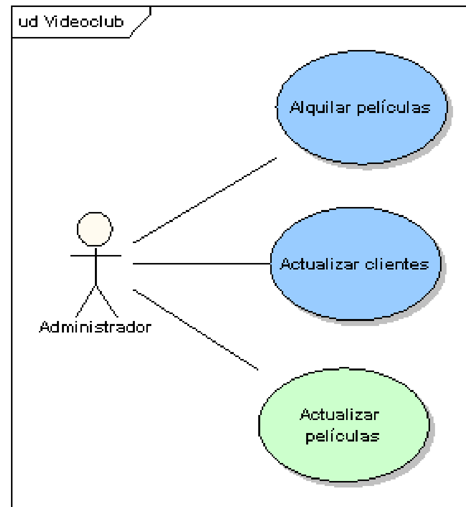
[9] REDBOOKS : TRANSACTIONS IN J2EE: Web oficial de IBM http://www.redbooks.ibm.com/abstracts/redp3659.html

[10] SUN DEVELOPER NETWORK : JSP ARCHITECTURE: Web oficial de Sun http://java.sun.com/developer/Books/javaserverpages/Chap12.pdf

[11] SUN DEVELOPER NETWORK : SERVLETS AND JSP PAGES BEST PRACTICES: Web oficial de Sun http://java.sun.com/developer/technicalArticles/javaserverpages/servlets_jsp/

A. Videoclub: Enunciado

Enunciado Queremos modelar un sistema para un videoclub que alquila películas. Los casos de uso que tenemos son:



Consideramos que el cliente siempre va a alquilar la película al local, entonces el usuario principal de nuestra aplicación es el encargado del videoclub a quien consideramos Administrador.

El Administrador actualiza los clientes (registra los nuevos y modifica los datos de los existentes). Como está suscripto a la Red Argentina del Videoclub, periódicamente recibe un archivo de novedades de películas y hace una importación de dicho archivo en el sistema del Videoclub.

Se pide que desarrolle los casos de uso "Alquilar películas" y "Actualizar clientes", teniendo en cuenta las siguientes definiciones en la interfaz de usuario:

A.1. Actualizar clientes

A.1.1. Pantalla principal

La pantalla tiene el siguiente formato:

Actualización de Clientes

Nombre contiene:

Dirección contiene:

Buscar Limpiar Nuevo

Cliente	Dirección			
				Alquilar
				Alquilar
				Alquilar
				Alquilar

Nuevo cliente

1. El Administrador elige la opción "Nuevo" y pasa la pantalla 1.1 en modo "Alta".

Modificación/Eliminación de Cliente/Alquilar películas

1. El Administrador puede buscar la lista de clientes por nombre o dirección o bien dejar los campos en blanco y elegir buscar.

2. El sistema devuelve los clientes que cumplen que el nombre o la dirección estén contenidos en el nombre/dirección del cliente. *Ejemplo*: si se busca "azca" en la dirección, devolverá un cliente que viva en la calle Nazca.

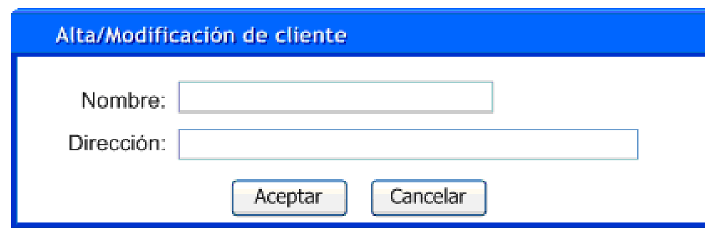
3. El Administrador selecciona un cliente de la lista y puede:

   a) modificarlo. Para modificar los datos del cliente, pasamos a la pantalla Edición de Cliente en modo "Modificación".

   b) eliminarlo. Para poder eliminar un cliente no debe tener pedidos asociados. Si no tiene pedidos, se elimina de la lista de clientes del videoclub.

4. En caso de querer generar un nuevo pedido para ese cliente, puede seleccionar la opción Alquilar (pasa al caso de uso "Alquilar películas").

Editar cliente



En esta pantalla se ingresarán o modificarán los siguientes datos de un cliente:

- el nombre
- la dirección

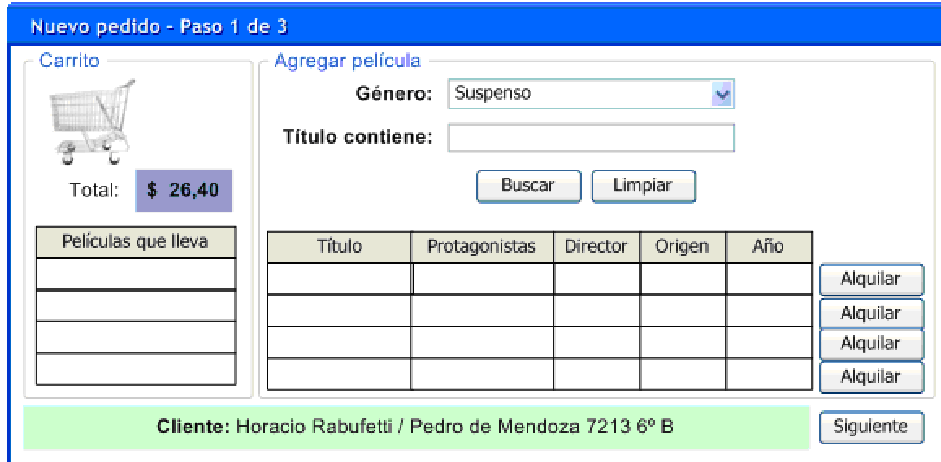
La única restricción es que los campos no pueden quedar en blanco. Al aceptar se actualizará la información del cliente.

A.2. Alquilar Películas

El sistema guiará al Administrador por tres pasos para asistirlo en la generación de un pedido. Un pedido está conformado de:

- varias películas (si quiere alquilar dos veces "Duro de matar 4" debe ingresar dos veces la misma película)
- la fecha del pedido
- el medio de pago (que puede ser Efectivo, Tarjeta de Crédito o Ticket Canasta).

Paso 1



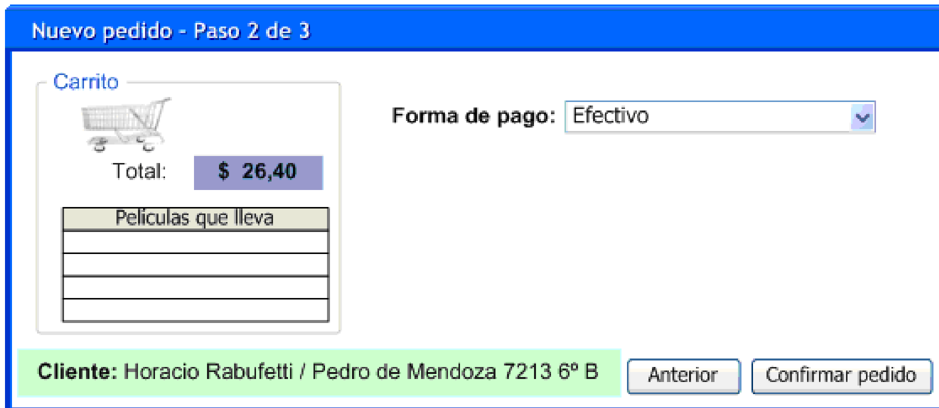
1. En esta pantalla el Administrador podrá seleccionar películas por Género exacto o un título que contenga una o varias palabras que él busque. Ejemplo: si busca "de matar" le aparecerá "Duro de matar 1", "Duro de matar 2", "Duro de matar 3", "Duro de matar 4". Luego selecciona la opción Buscar.

2. El Sistema devolverá la lista de películas encontradas.

3. El Administrador podrá seleccionar la película y agregarla al carrito de películas que lleva ("Alquilar"). El sistema automáticamente irá calculando el importe. El cliente puede seleccionar muchas veces la misma película si así lo desea, se llevará n copias de la misma película.

4. El Administrador puede seguir buscando más películas hasta que decida elegir la opción "Siguiente". No debe permitir pasar a la siguiente pantalla si no seleccionó ninguna película.

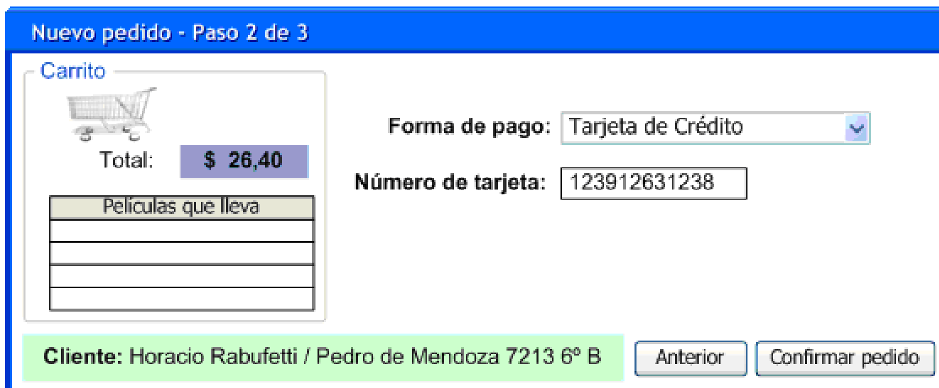
Paso 2



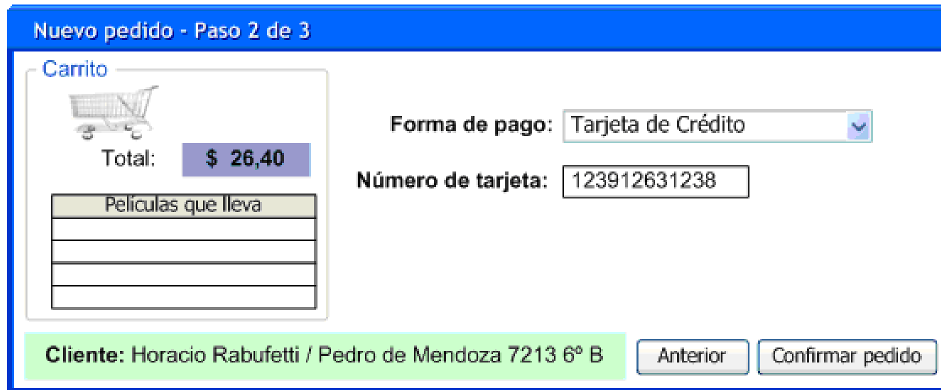
1. En esta pantalla el Administrador podrá seleccionar el medio de pago con el que abona el cliente: por el momento se prevé Efectivo, Tarjeta de Crédito y Ticket Canasta. Considerar que todas las películas tienen un valor, por lo que el total siempre será mayor a cero.

2. Para confirmar el pedido debe seleccionarse algún medio de pago.

Efectivo y Ticket Canasta no requieren información adicional, mientras que cuando se ingrese la Tarjeta de Crédito se pedirá el número de tarjeta:



Paso 3



Se generará un nuevo pedido asociándolo al cliente. La pantalla mostrará la información del pedido y un mensaje indicando el resultado de la operación (ok - error).