Resumen de clase
Ejemplos creacionales

Ideas de Diseño sobre ejercicios anteriores

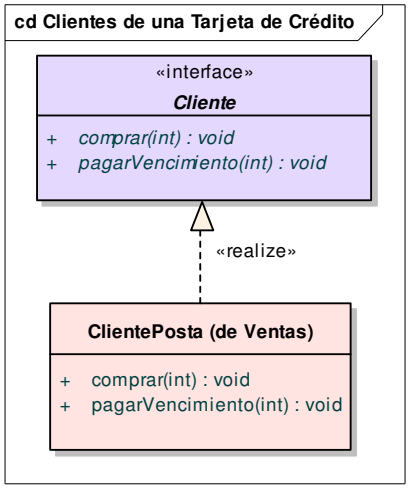
1° cuatrimestre 2010

Contenido

Ejemplo Clientes de una Tarjeta de Crédito: Enunciado

Pertenecemos a la gerencia de Condiciones Comerciales de una empresa emisora de una Tarjeta de Crédito. La gerencia de Ventas nos provee una interfaz Cliente, cuyos contratos son:

- comprar(int monto)
- pagarVencimiento(int monto)



Se pide contemplar los siguientes requerimientos:

- algunos clientes adheridos a una promoción suman 15 puntos por cada compra mayor a $ 50.
- además, algunos clientes contrataron el sistema 'Safe Shop', que bloquea compras de la tarjeta mayores a un monto que el cliente fija.

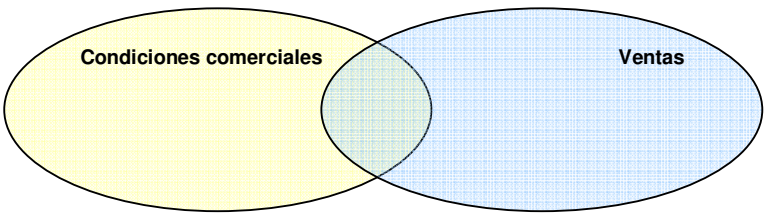
Sobre el dominio

Tenemos dos sectores dentro de la empresa de tarjetas de crédito:

- Ventas
- Condiciones Comerciales

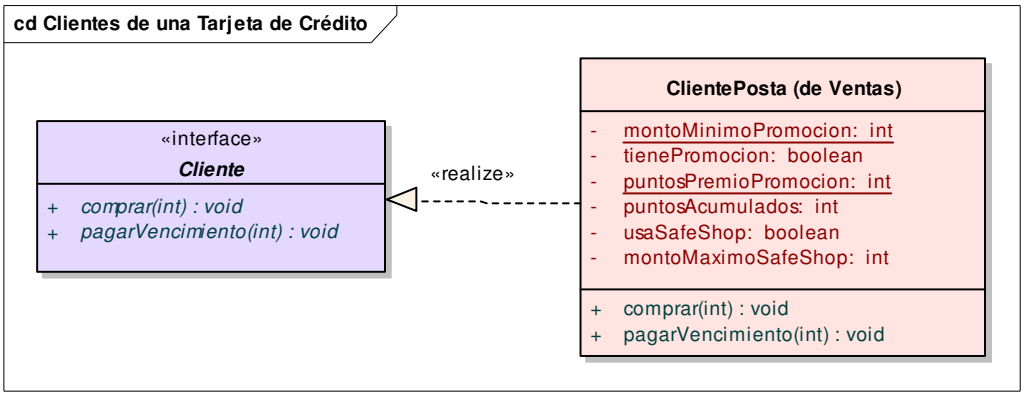
¿Qué responsabilidades cumplen cada una?
Si bien la empresa tiene vendedores que son quienes interactúan en muchos casos con los clientes (para ofrecer nuevos servicios), en muchas empresas existe un sector que determina restricciones o acuerdos que los vendedores deben cumplir, que son las “condiciones comerciales”.
Ejemplos:

- Todos los clientes del exterior tienen un descuento por pronto pago del 20%
- Sólo se puede trabajar con clientes mayoristas
- Los clientes Rabufetti y Conesta aceptan cheques a 30/60 y 90 días como forma de pago



Vemos que Cliente intersecta el negocio de ambos sectores (Ventas y Condiciones Comerciales). Ahora veremos cómo atacar estos requerimientos.

Solución 1: Una única clase



¿Cómo creamos un Cliente con Safe shop?

a) Con booleanos

```
Cliente cliente = new Cliente(true, false);
```

Desventaja: no queda claro cuando lo escribo que el cliente tiene safe shop. Me puedo confundir fácilmente al invertir los booleanos…

b) Defino constantes

```
public final boolean CON_SAFE_SHOP = true;
public final boolean SIN_SAFE_SHOP = false;

Cliente cliente = new Cliente(CON_SAFE_SHOP, SIN_PROMOCION);
```

Aun así, podría invertir los parámetros y no darme cuenta…

c) Defino enums:

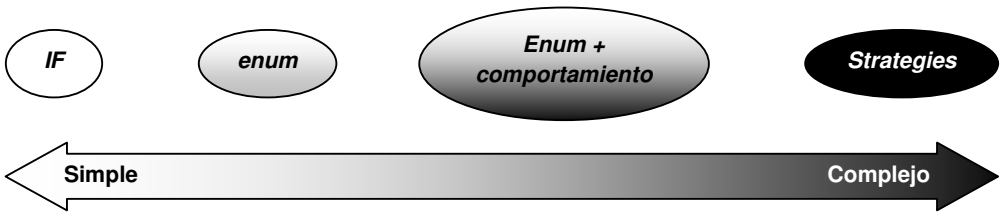
```
public enum SafeShop {

    CON() { public boolean tiene() { return true; } },
    SIN() { public boolean tiene() { return false; } };

    public boolean tiene() {
          return tiene();
    }

}

…

Cliente cliente = new Cliente(SafeShop.CON, Promocion.SIN);
```

Esto clarifica que el primer parámetro define si el cliente tiene o no safe shop y el segundo tiene o no promoción. El enum actúa como un objeto con poco comportamiento: en definitiva podríamos pensar en tratar de delegar más al enum en lugar de usarlo para guardar un true o

un false: definimos el método comprar() en cada enum agregando cada uno de los comportamientos.

Para implementar strategies bien conocidos y donde el código no sea muy complejo, quizás convenga usar enums como strategies en lugar de usar varias clases que implementen una interfaz (o bien varias subclases contra una superclase). La contra de este approach es que los enums no escalan cuando la solución se vuelve compleja.

En general, para modelar una decisión tenemos rangos de soluciones que abarcan desde lo más simple a lo más complejo:



d) Podemos usar un factory method: un método que instancia un cliente con Safe Shop. Este método puede estar en un objeto que trabaje como Factory o bien dentro de cliente como método estático:

```
ClientePosta.crearSafeShop();
```

Ventaja: ya no se cómo se implementa internamente. Sí se que el cliente tiene safe shop, porque *es un concepto que el negocio entiende*. Está bien que yo sepa que el cliente tiene safe shop, lo que no quiero saber es que el cliente tiene un booleano para representarlo (porque si el cliente decide cambiar de boolean a int yo me veo afectado).

Desventaja: en un esquema con muchas combinaciones tengo demasiados métodos (cliente sólo con safe shop, cliente sólo con promoción, cliente con safe shop y promoción, etc.)

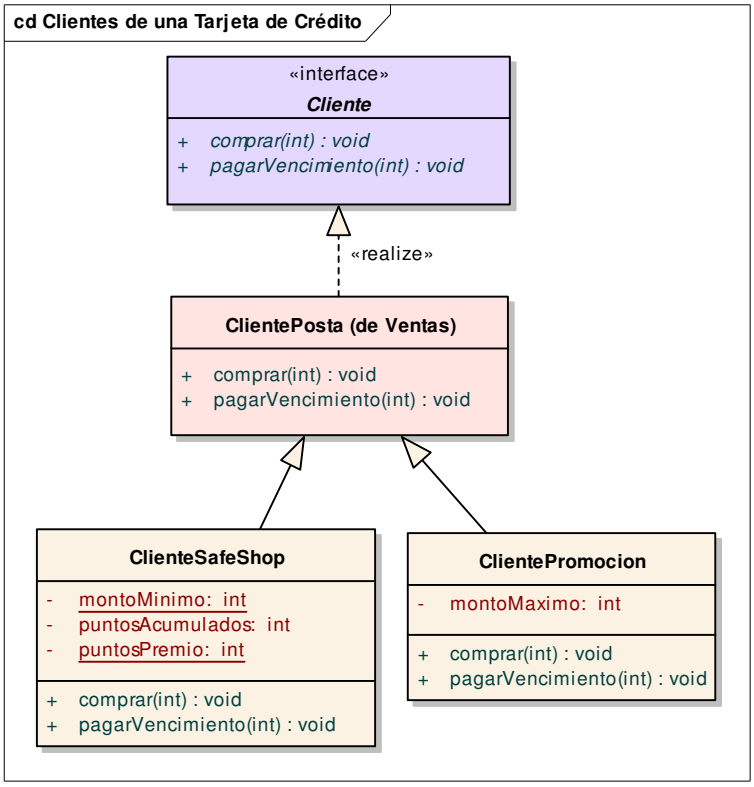
e) La alternativa tradicional es tener setters para cada propiedad:

```
Cliente cliente = new Cliente();
cliente.setSafeShop(true);
```

Ventaja: calzan bien las combinaciones y queda claro lo que estoy haciendo.

Desventajas: instanciar el cliente dejó de ser una operación atómica (hay un momento en el que el estado del cliente quedó inconsistente). También necesito muchas líneas para decir que el cliente tiene safe shop.

Solución 2: Subclasificar



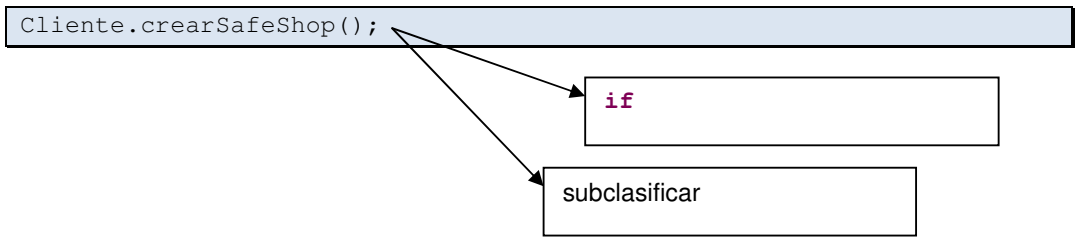
Ya hemos visto que esta solución no era adecuada, pero a fines didácticos nos interesa pensar que podría llegar a funcionar. Entonces qué alternativas tenemos:

a) Si tenemos un cliente con safe shop, es un new ClienteSafeShop(). Como el new en Java es una palabra reservada, no puedo redefinir el new, pero sí puedo trabajar con un factory method: puede ser un objeto aparte o bien un método estático en Cliente:

```
Cliente.crearSafeShop();
```

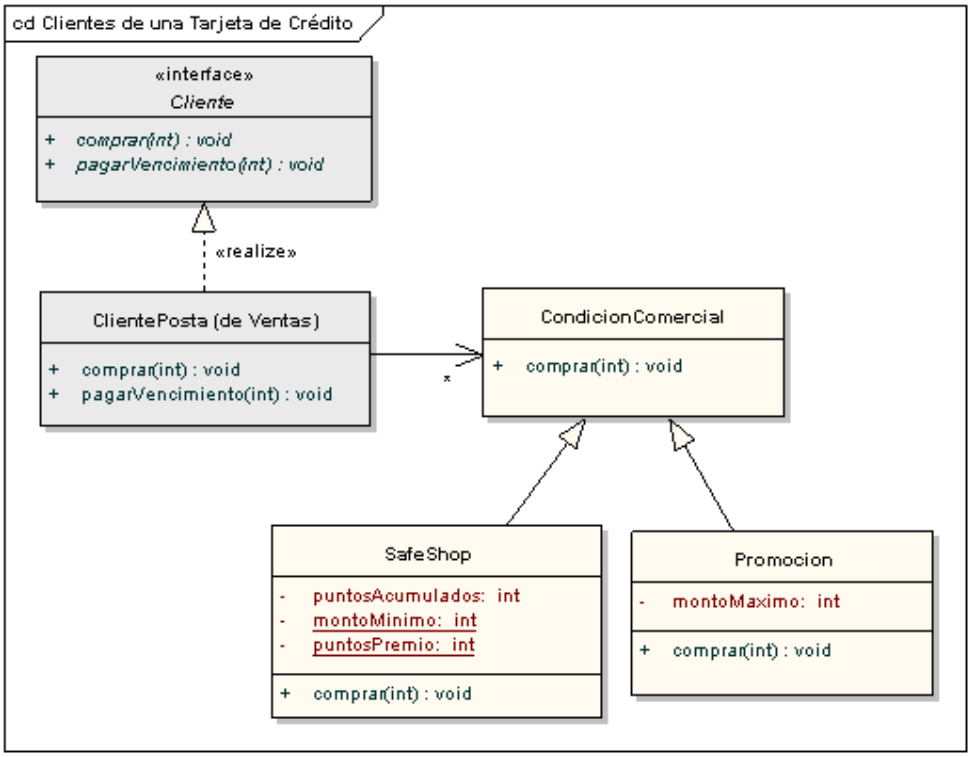
¡ésta era la solución 1.d)!

Entonces tengo una ventaja a favor del factory method, puedo pasar de la solución 1) a la 2) sin que los que instancien a los clientes se vean afectados: esto lo convierte en una buena herramienta para encapsular.



Fíjense que ninguna de las otras opciones me permite pasar de la solución 1) a la 2), eso no las invalida pero plantea una ventaja diferencial que por ahí antes no lo tenía en cuenta.

Solución 3: Tener una colección de condiciones comerciales



a) Podemos construir los strategies con un Builder:

```
ClienteBuilder builder = new ClienteBuilder();
builder.addSafeShop()
Cliente cliente = builder.build();
```

El constructor ClienteBuilder instancia un cliente posta:

```
public ClienteBuilder() {
   this.cliente = new ClientePosta();
}
```

Mostramos cómo agregar la funcionalidad Safe Shop:

```
public void addSafeShop() {
   this.cliente.agregarCondicionComercial(new SafeShop());
}
```

El build devuelve el cliente que tenga el Builder:

```
public Cliente build() {
   return this.cliente;
}
```

Incluso podríamos mejorar la legibilidad de este código, permitiendo encadenar los mensajes del Builder:

```
Cliente cliente = new ClienteBuilder()
                        .addSafeShop()
                        .build();
```

Para eso el método addSafeShop() debe devolver el mismo builder (y en general todos los métodos que le configuran cosas al Builder):

```
public ClienteBuilder addSafeShop() {
   this.cliente.agregarCondicionComercial(new SafeShop());
   return this;
}
```

El addSafeShop() es cómodo cuando se trabaja con el Builder en un test o en un objeto que instancia clientes. Sin embargo si modeláramos la interfaz de usuario, seguramente tendríamos algo como:

Entonces quizás sea más cómodo aquí definir un método setSafeShop que acepte el valor booleano que el usuario tilde:

```
Cliente cliente = new ClienteBuilder()
                        .setSafeShop(booleano)
                        .build();
```

b) Otra variante es hablar directamente con el cliente, pasándole un Strategy:

```
Cliente cliente = new Cliente(new SafeShop());
```

¿Qué pasa si quiero enviarle al cliente muchos strategies? A partir de Java 1.5 tengo la posibilidad de mandar un conjunto de parámetros variables:

```
public Cliente(ClienteStrategy... criterio) {
```

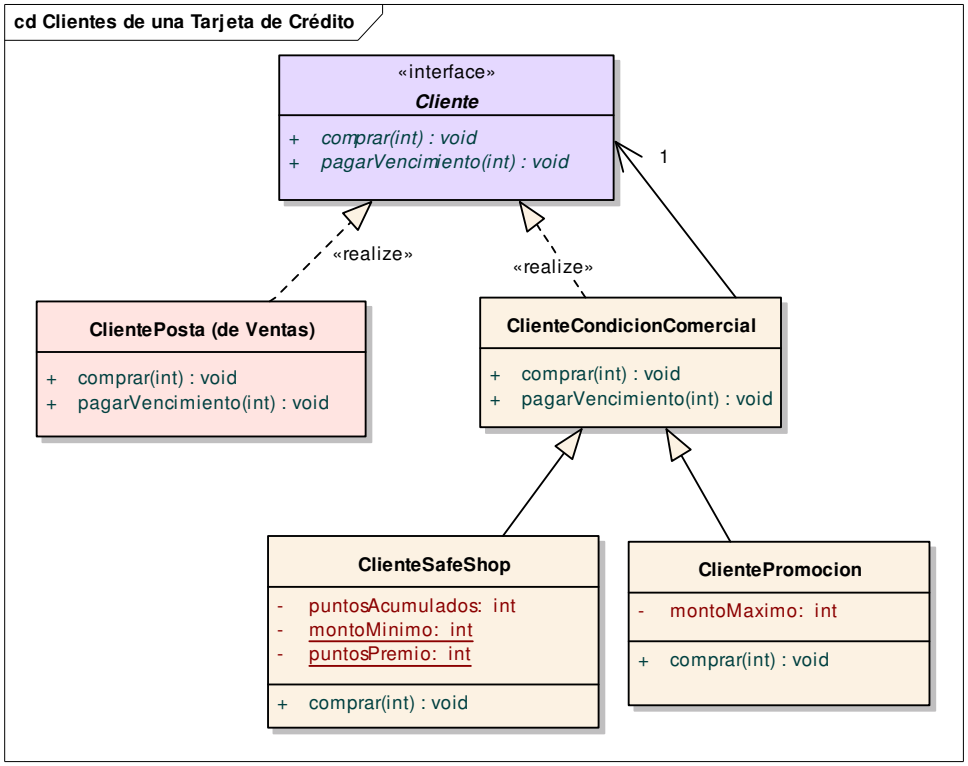
De esa manera puedo crear un cliente pasándole la cantidad de strategies que precise:

```
Cliente cliente = new Cliente(new SafeShop(), new Promocion());
```

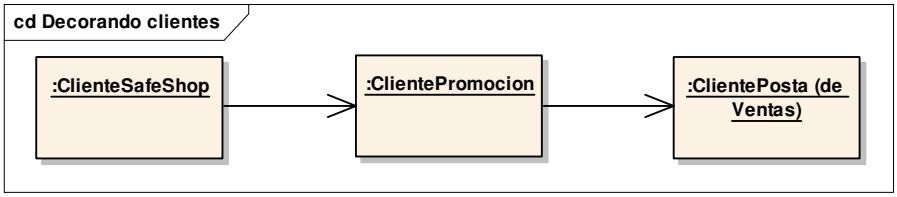
c) También podemos agregar manualmente la colección de strategies:

```
Cliente cliente = new Cliente();
cliente.addSafeShop();                  → cuando son pocos y conocidos
cliente.addCondicion(new Promocion());→ esta opción es más abierta
```

Solución 4: Decorar condiciones comerciales



Para instanciar un safe shop, primero tenemos que instanciar un cliente:



Como el cliente no sabe que lo decoran, tenemos que trabajar sobre el constructor de Safe shop, no sobre el constructor del cliente. Entonces veamos qué opciones tenemos:

a) En el constructor de safe shop aceptamos un objeto que implementa la interfaz cliente (puede ser otro decorador o el decorado final: el cliente posta)

```
new SafeShop(new Promocion(new Cliente()));
```

b) Usamos un Builder:

```
Cliente cliente = new ClienteBuilder()
                        .addSafeShop()
                        .build();
```

Nuevamente, recalcamos que usando builders no nos importa cómo se implementen las condiciones comerciales, el código de quien instancia clientes no se ve afectado. El Builder sirve inclusive si decidimos aplicar la solución 5) –decorar los strategies-.

Mostramos un poco cómo se implementan los métodos de ClienteBuilder:

```
public ClienteBuilder() {
   this.cliente = new ClientePosta(); ← igual que en (3)
}

public void addSafeShop() {
   this.cliente = new SafeShop(this.cliente);
   return this;
}

public Cliente build() {
   return this.cliente;              ← igual que en (3)
}
```

Parte 2: Enunciado Ejercicio Firewall

Se requiere modelar un firewall, que actúa en el punto de entrada a una LAN. Se controlarán sólo las comunicaciones entrantes, procedentes de una WAN.
El firewall recibe mensajes de la red WAN y las forwardea solamente a la LAN (o sea, que sólo maneja mensajes entrantes).

Debe permitir las siguientes funcionalidades:

Configuración y ejecución de reglas

Control de acceso a puertos: debe permitir habilitar y/o bloquear puertos individualmente y por conjuntos.

Ejemplo: Todos los puertos inferiores al 1024 están bloqueados, todos los superiores están habilitados salvo el 8080. Además, los puertos 80, 23, 25 y 110 deben estar habilitados.

Filtrado de IPs: debe permitir filtrar la IP o hostname de destino y de origen, tanto individualmente como por rangos.

Ejemplos:

- No se aceptará ningún mensaje proveniente de la IP 24.35.126.155.
- La IP 192.168.0.19 no podrá recibir ningún mensaje proveniente de la 24.56.178.165.
- Las IP desde la 192.168.0.1 hasta la 192.168.0.10 no pueden recibir mensajes de la 24.35.126.154.

Acción ante el bloqueo: Si se bloquea un mensaje de entrada, se puede tomar la decisión de enviar un mensaje a la IP origen informando que el envío no fue exitoso. También se puede tomar la decisión de loguear (Inicialmente pensemos en informar por consola) que el mensaje fue bloqueado.

Ejemplos:

- No se aceptará ningún mensaje proveniente de la IP 24.35.126.154 y se loguearán todos los mensajes recibidos desde esa IP.
- La IP 192.168.0.19 no podrá recibir ningún mensaje y se debe informar al origen que se denegó la entrega del mensaje.

Otras acciones:

Se puede decidir loguear un mensaje a enviar, aún cuando se permita enviarlo.

Ejemplo:

- Todos los mensajes al puerto 1521 deben loguearse.

Se puede decidir realizar un forward de una IP/puerto

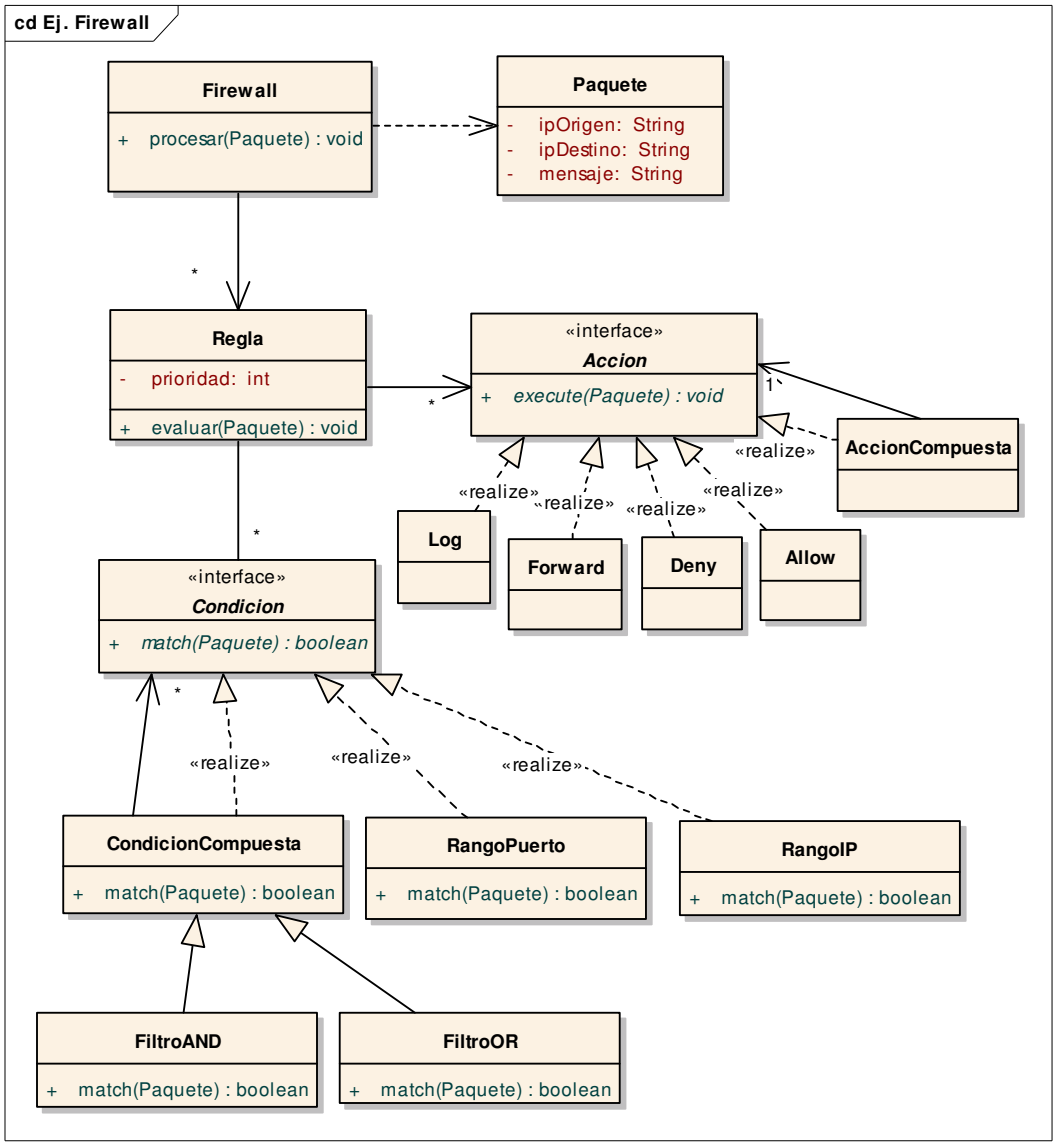
Ejemplo:

- Los mensajes a la IP 192.168.0.1 puerto 80, deben ser redirigidos a la IP 192.168.0.4 puerto 8866.

Si bien no se considerarán en esta iteración, en futuras iteraciones aparecerán nuevas posibles acciones.

Una solución posible

Dejamos aquí el diagrama de clases con una solución posible:



¿Cómo creo un firewall?

```
Firewall firewall = new Firewall();
```

Carga de acciones compuestas

Ok, la complejidad surge cuando tengo que crear una regla. Pensando primero en las acciones compuestas, no puedo combinar acciones excluyentes entre sí: allow y deny no son válidas juntas. ¿Cómo hago para poner eso en algún lado?

Yo podría elegir que el usuario lo configure bien, pero elegimos validarlo nosotros.

Una opción posible: tengo 4 acciones posibles para combinar en una colección. Cuando el usuario elige deny, entonces saco allow, forward y deny de la colección de acciones posibles (sólo puedo agregar log). Ese comportamiento va en la clase Deny. Pero lo mismo hay que hacer en cada una de las acciones (lo cual se vuelve un poco engorroso).

Otra opción: hay acciones que pueden aceptar combinaciones (log) y otras que no ➔ **Reifico**[1] una idea para que no esté metida dentro del código. Así le ponemos nombre a ciertas acciones que son *terminales/terminantes/excluyentes*: allow, deny, forward y las discriminamos de las acciones que no lo son: log. Para eso me puede ayudar el orden: primero agrego las acciones que no son excluyentes y por último las que son excluyentes.

Vamos a utilizar un Builder para construir una regla:

```
Regla r = new Regla (new AccionBuilder()
                            .log(-- parámetros --)
                            .deny().
                            .build(),
                        -- la condición la trataremos después -- );
```

Ahora nos metemos en el código de ese Builder. Internamente trabaja con una colección de acciones:

```
public AccionBuilder addAccion(Accion a) {
    acciones.add(a);
    return this;
}

public AccionBuilder log(-- parámetros --) {
    return this.addAccion(new Log(-- parámetros --));
}
```

Estos dos métodos trabajan en distintos niveles:

1) el addAccion es de más bajo nivel, permite pasarle cualquier tipo de acción al builder.
2) El log es de más alto nivel (trabaja sobre un tipo de acción conocida para el usuario), termina delegando al método de más bajo nivel.

Ahora vamos a agregar la validación de que la acción se pueda agregar o no: cambiamos la jerarquía de Accion para crear una superclase nueva AccionExcluyente. Lo siguiente es modificar el addAccion del AccionBuilder:

[1] **Reificar**: tratar un proceso o una propiedad como si fuera una cosa. En el contexto del diseño es encontrar una abstracción y darle entidad (a través de un objeto, una clase o un método)

```
public AccionBuilder addAccion(Accion a) {
    for (Accion accion : acciones) {
        if (!accion.sosCompatibleCon(a)) {
            throw new BusinessException("No puede agregar la acción "
+ a + " porque hay acciones excluyentes entre sí");
        }
    }
    acciones.add(a);
    return this;
}
```

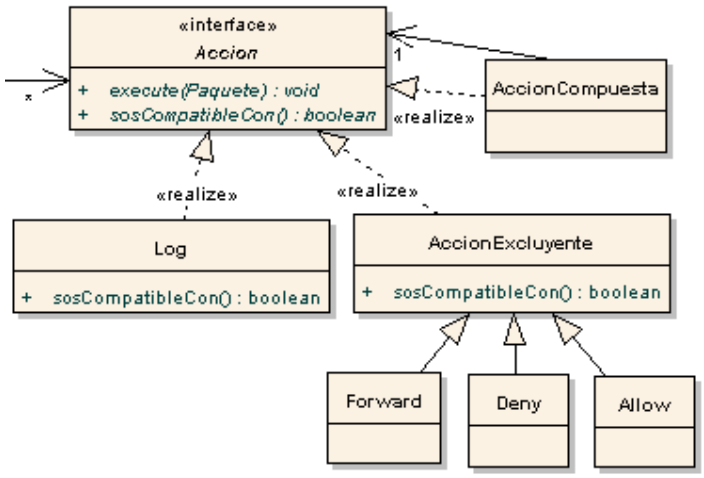
Veamos cómo se implementa el sosCompatible en Log:

```
public boolean sosCompatibleCon(Accion a) {
    return true;
}
```

El sosCompatible en Excluyente se implementa así:

```
public boolean sosCompatibleCon(Accion a) {
    return (!a.sosExcluyente());
}
```

sosExcluyente() devuelve true para las acciones excluyentes y false para Log.



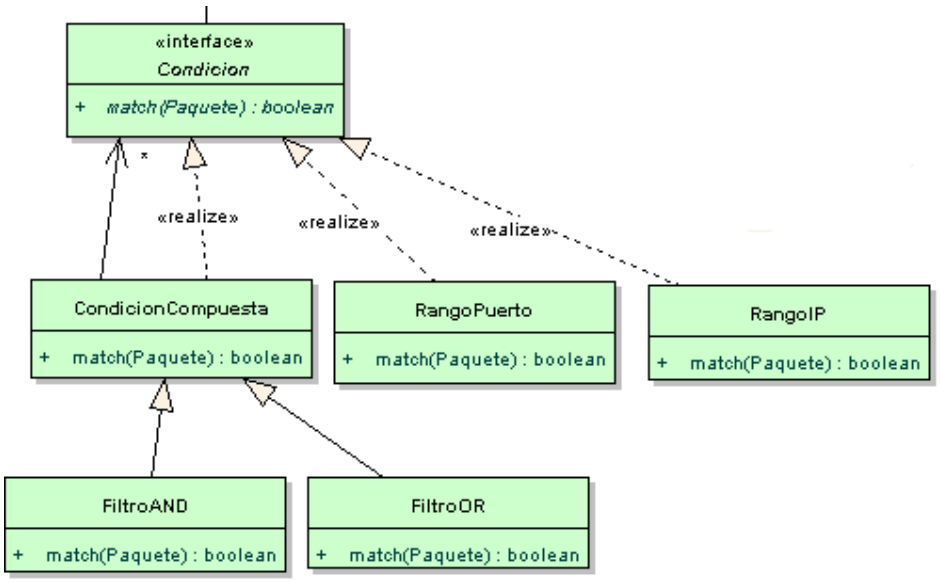
El build del AccionBuilder también cambia:

```
public Accion build() {
     if (this.acciones.isEmpty()) {
         throw new BusinessException("Falta definir al menos una
acción");
     }
     if (this.acciones.size() == 1) {
         return acciones.get(0);
```

```
        }
        return new AccionCompuesta(this.acciones);
}
```

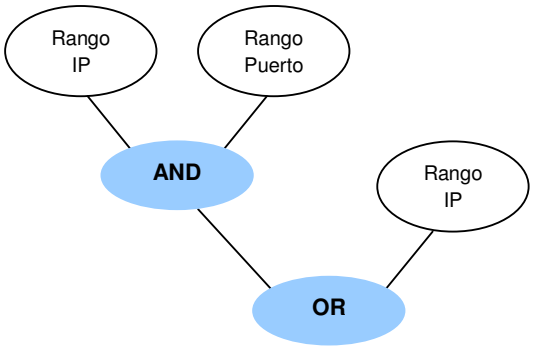
Condiciones compuestas con AND y OR

Uno de los desafíos que tenemos con el tema de las condiciones es que hay que operarlas mediante ands y ors:



Aquí nuestra idea es prescindir del Builder, y tratar de implementar un factory method en la clase Condicion:

```
Regla r = new Regla (new AccionBuilder()
                            .log(-- parámetros --)
                            .deny().
                            .build(),
                     new RangoIP(-- parámetros --)
                            .and(new RangoPuerto(-- parámetros --))
                            .or(new RangoIP(-- parámetros --))
          );
```



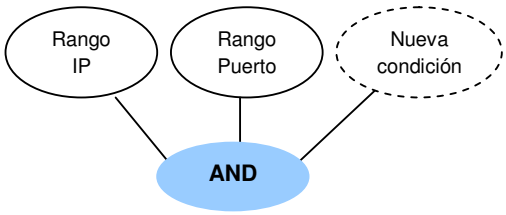
Vamos a implementarlo: como método de instancia de la clase Condicion tenemos

```
public Condicion and(Condicion otra) {
    return new And(this, otra);
}
```

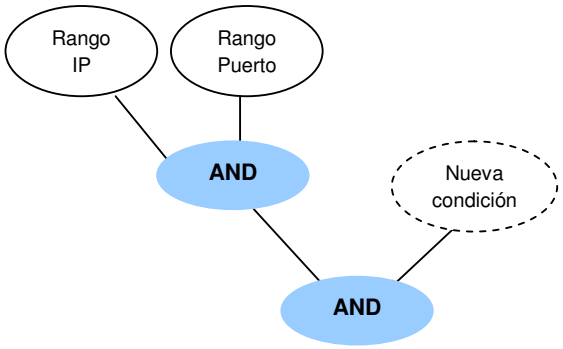
Agregamos un chiche: en la clase And redefinimos el comportamiento para que no devuelva otra instancia de And: agrega una condición más a la colección (ojo, que el grafo de arriba puede confundir, el operador and se aplica sobre *muchas* condiciones, no solamente sobre dos):

```
public Condicion and(Condicion otra) {
    this.hijas.add(otra);
    return this;
}
```

Y tendría este efecto:



En lugar de:



De todos modos, es un detalle.

Volviendo sobre las acciones

Podríamos aplicar la misma idea para generar acciones compuestas, con un factory method de instancia en Accion:

```
            new Regla(new Log()
                        .combinarCon(new Deny()
            -- instanciación de la condición --
            );
```

Resolvemos el método combinarCon de Accion:

```
public Accion combinarCon(Accion otra) {
    return new AccionCompuesta(this, otra);
}
```

En la clase AccionExcluyente lo redefinimos para preguntar si ya hay una regla excluyente:

```
public Accion combinarCon(Accion otra) {
    if (otra.sosExcluyente()) {
       throw new BusinessException("No puede agregar la acción " + a
+ " porque hay acciones excluyentes entre sí");
    } else {
       return super.combinarCon(otra);
    }
}
```

Aparecen los lenguajes

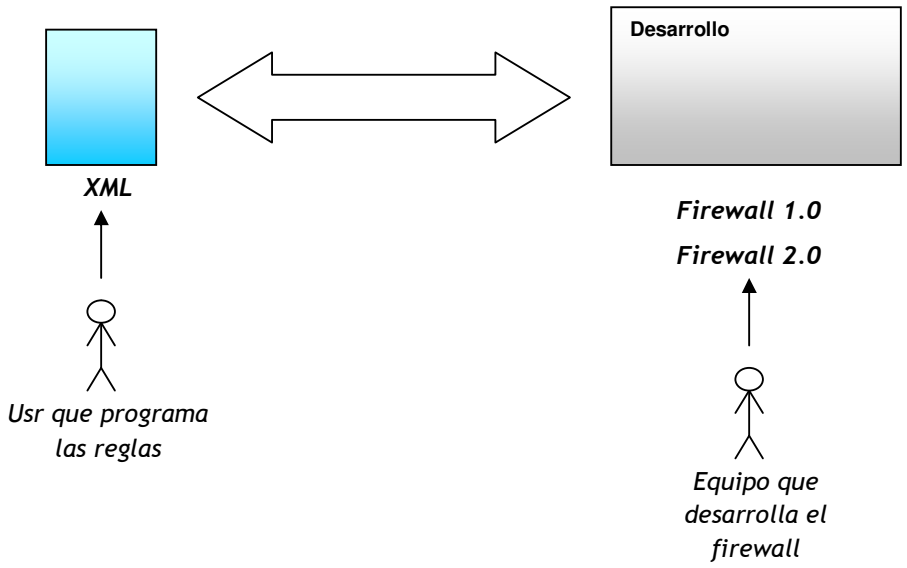
En definitiva, el and, el or y el combinar son mensajes que me llevan a crear un lenguaje que cada vez se parece más al lenguaje de dominio. Otra alternativa para configurar el firewall podría ser crear un XML, un archivo con tags que luego pueda ser interpretado (parseado) por un objeto que genere las instancias correspondientes:

```
<regla nombre="test">
    <acciones>
        <log  -- parámetros --/>
        <deny -- parámetros --/>
```

```
    </acciones>
    <or>
        <and>
            <rango-puerto -- parámetros --/>
            <rango-ip     -- parámetros --/>
        </and>
        <rango-puerto -- parámetros --/>
    </or>
</regla>
```

Y a medida que subo de nivel, construyo cosas más cercanas al dominio y digo menos cómo hacerlo ➔ voy siendo más **declarativo**

Mientras no se agreguen features nuevos la compatibilidad del xml se mantiene:



En resumen, al pensar en la instanciación de los objetos vamos generando herramientas –dentro y fuera de Java- que van subiendo el nivel de abstracción, así

- Simplificamos la creación de los objetos
- Nos acercamos a un lenguaje mucho más fácil de comprender para el usuario