Patrones de Diseño

EJERCICIOS

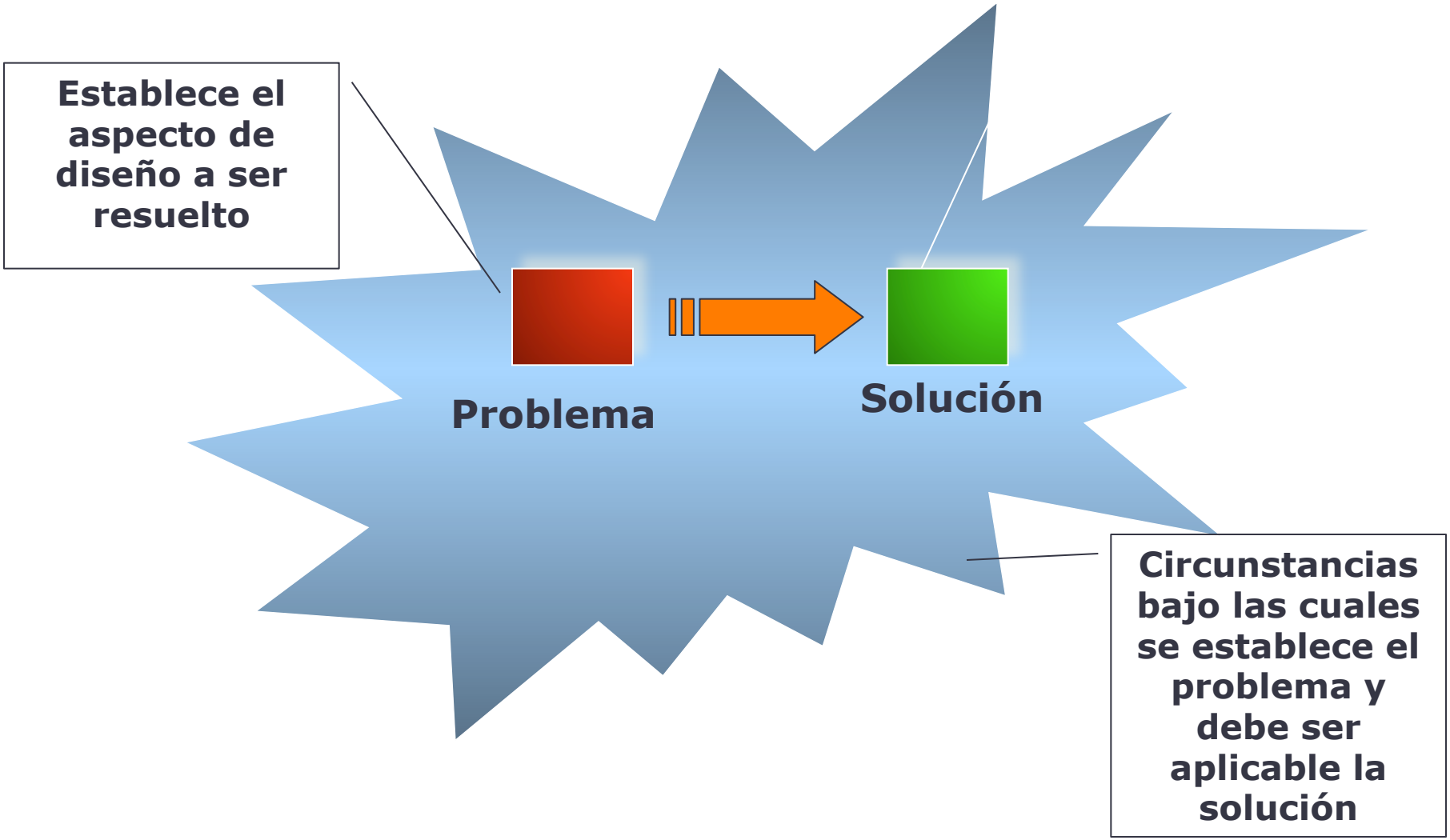
Ingeniería del Software I

Carlos Blanco
Universidad de Cantabria

Introducción

- Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo
- Propuestos por el Gang of Four (Gamma, Helm, Johnson y Vlissides).
  - *Design Patterns. Elements of Reusable Object-Oriented Software* - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison Wesley (GoF- Gang of Four). Addison Wesley. 1994.
- Son un esqueleto básico que cada diseñador adapta a las peculiaridades de su aplicación.

Introducción



Clasificación

		Ámbito	
		Clase	**Objeto**
Propósito	**Creacional**	Método Fábrica	Fábrica Abstracta, Constructor, Prototipo, Singleton
	Estructural		Adaptador, Puente, Compuesto, Decorador, Fachada, Peso Mosca, Apoderado
	De Compor-tamiento	Intérprete, Método Plantilla	Cadena de Responsabilidad, Comando, Iterador, Mediador, Memento Observador, Estado Estrategia, Visitante

Problema

- En un sistema bancario existen diferentes familias de productos con características distintas.
- Están asociados a varios tipos de cuenta que dependen del tipo de cliente que las abra:
  - Cuenta Joven, para clientes < 25 años
  - Cuenta 10, para clientes entre 26 y 65 años y con nómina domiciliada
  - Cuenta Oro, para mayores de 65 años con pensión
  - Cuenta Estándar, para clientes que no encajan en las anteriores

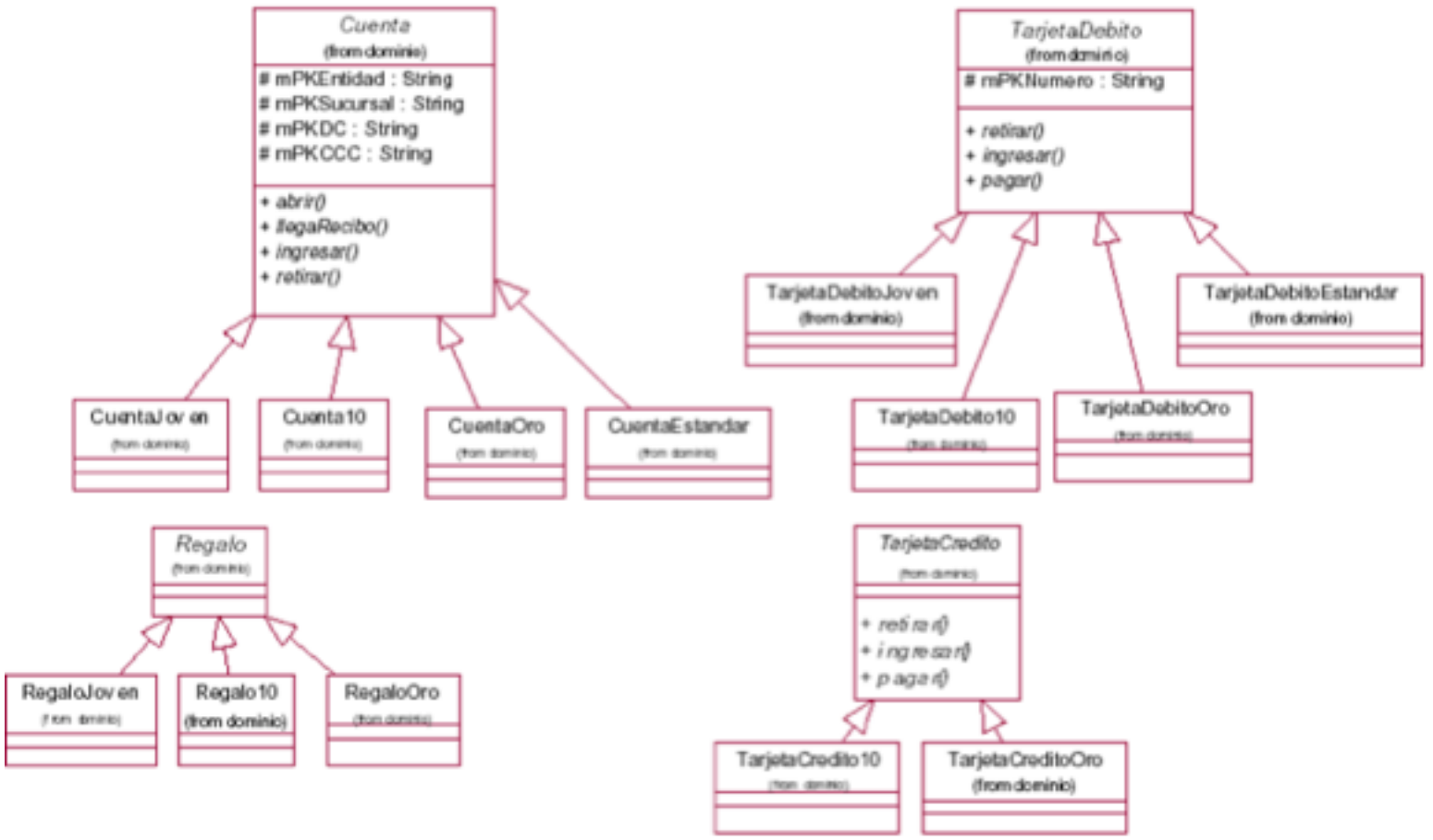
Problema

- Las características de los productos se resumen en la siguiente tabla:

Tipo de cuenta	Cuenta	Tarjeta débito	Tarjeta crédito	Regalo
Joven	2% de interés	Gratuita	No	CD música
10	1% de interés 50% descubierto	Gratuita	18 euros 60% nómina	Reproductor CD
Oro	1,5%	Gratuita	Gratuita 60% pensión	Seguro
Estándar	0,5%	5 euros	No	No

- Tal y como se ve, aunque puede parecer que se ofrecen siempre los mismos productos, existen ciertas particularidades entre ellos que hacen que sean distintos, ciertas características que hacen que se construyan de manera distinta.
- **Representar el conjunto de productos mediante un diagrama de clases**

- Diagrama de clases para el conjunto de productos:



- Podríamos crear uno a uno los elementos correspondientes a cada uno de los tipos de cuenta... pero no es elegante
- Nos gustaría tener la opción de que al crear una cierta cuenta se crearan automáticamente los productos asociados al tipo de cuenta

¿ Cómo podríamos hacerlo ?

Fábrica Abstracta

ABSTRACT FACTORY

- **Especifica como crear familias de objetos que guardan cierta relación entre sí.**

- Nosotros tan sólo tenemos que utilizar las **interfaces** que se nos proponen para interactuar con el sistema, sin importar el tipo de elemento a instancias, ya que la fábrica lo hará por nosotros.
  - Se **generaliza** el **comportamiento** de una familia de clases.
  - Conseguimos que el sistema sea **independiente** de cómo se crean, componen o representan las entidades representadas.
  - Los sistemas se vuelven fácilmente **extensibles**.

Fábrica Abstracta

ABSTRACT FACTORY

- La fábrica abstracta tiene operaciones abstractas (por cada producto)
- Tenemos tantos tipos de fábricas concretas como de productos.
- Cada fábrica concreta conoce a los productos que puede fabricar.

[para simplificar sólo están indicadas las relaciones de FabricaJoven]

Fábrica Abstracta

ABSTRACT FACTORY

- **La creación de la familia de productos podría residir en el siguiente método**

```
public void crearLote(Cliente c) {
 Fabrica f;
 if (c.getTipo()==CLIENTE_JOVEN)
    f=new FabricaJoven(c);
 else if (c.getTipo()==CLIENTE_10)
    f=new Fabrica10();
 else if (c.getTipo()==CLIENTE_ORO)
    f=new FabricaOro(c);
 else
    f=new FabricaEstandar(c);
 f.crearCuenta();
 f.crearTDebito();
 f.crearTCredito();
 f.crearRegalo();
}
```

- En este código existe un objeto f de tipo Fabrica (que es una clase abstracta);
- Dependiendo del tipo de cliente, f se instancia al tipo de fábrica correspondiente.
- A continuación, se ejecutan las cuatro operaciones de creación del lote de productos. Estas operaciones son abstractas en Fabrica, pero concretas en sus subclases.
- Puesto que, al llegar a la sentencia f.crearCuenta() la variable f ya está instanciada a una fábrica concreta, se ejecuta la versión adecuada de la operación.

Problema

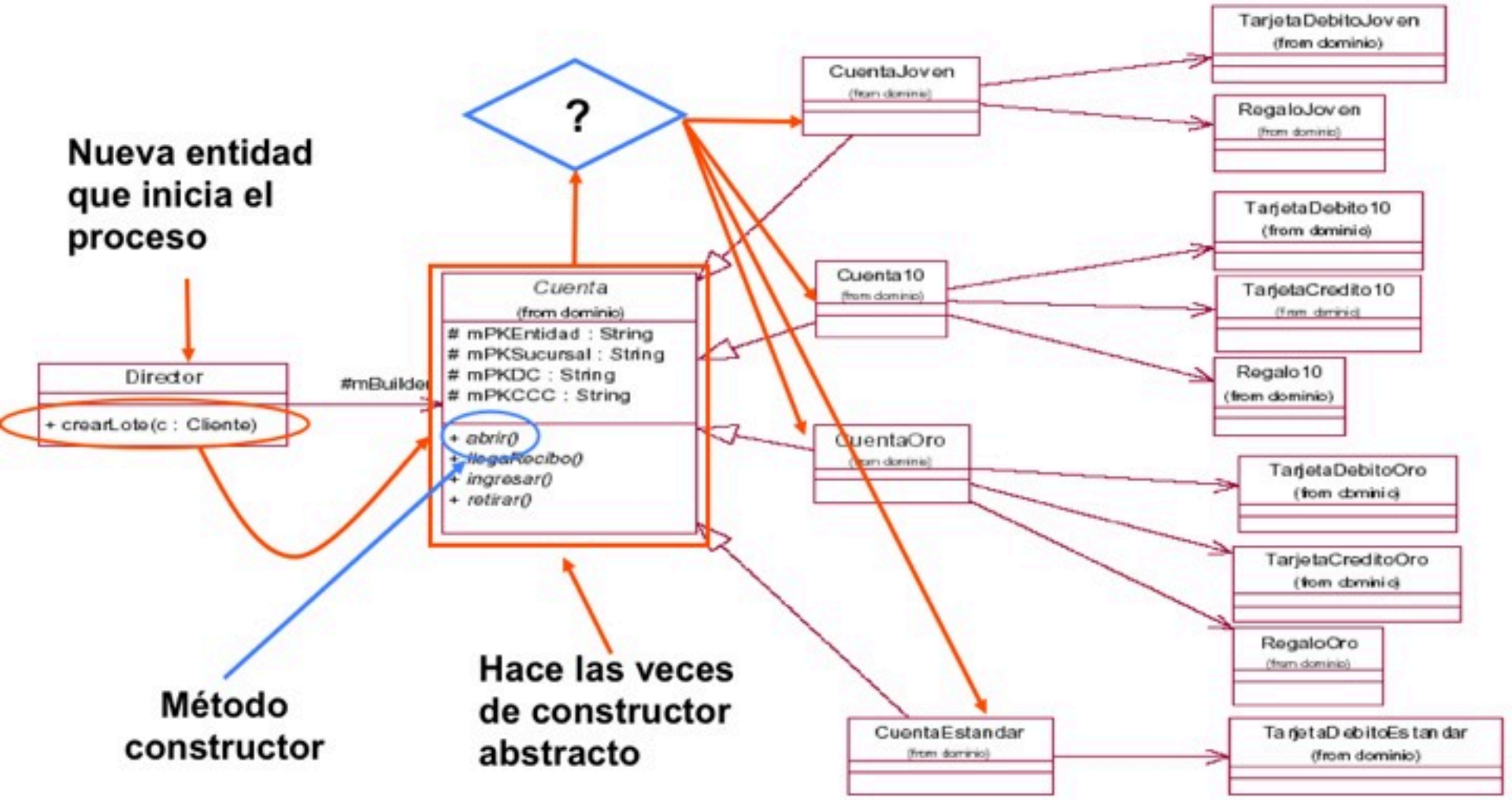
- Siguiendo con el ejemplo anterior...

- Queremos construir **objetos complejos** que pueden ser **de distinto tipo** (los tipos de cuenta).
- Utilizando para ello otros objetos y una secuencia de pasos definida (crear el tipo de tarjeta, regalos, etc.).

- Ahora nos interesaría:
  - Asegurarnos que se utiliza siempre el **mismo proceso** (la misma secuencia de pasos) para crear una cuenta de un determinado tipo o un lote formado por varias cuentas,...
  - La idea es buscar un desacople de las funciones del constructor (la clase Cuenta de la Fábrica Abstracta).

Constructor

BUILDER



Constructor

BUILDER

- En la figura anterior, la clase Cuenta hace las veces de constructor abstracto (de Builder abstracto).
- Obsérvese también que hemos añadido una nueva clase (Director), que es la encargada de llamar al constructor, una vez instanciado.
- Con esta estructura, la creación de la familia de productos podría residir en el siguiente método:

```
public class Director {
 Cuenta mBuilder;
 ...
  void crearLote(Cliente c) {
   if (c.getTipo()==CLIENTE_JOVEN)
      mBuilder=new CuentaJoven(c);
   else if (c.getTipo()==CLIENTE_10)
      mBuilder =new Cuenta10();
   else if (c.getTipo()==CLIENTE_ORO)
      mBuilder =new CuentaOro(c);
   else
      mBuilder =new CuentaEstandar(c);
   mBuilder.abrir();
  }
}
```

Constructor

BUILDER

- Cada constructor concreto (clases CuentaJoven, Cuenta10, CuentaOro y CuentaEstandar) es el responsable de crear los objetos correspondientes, lo cual se hace en el método abrir(), que recibe diferentes implementaciones en cada clase
- Ejemplo:

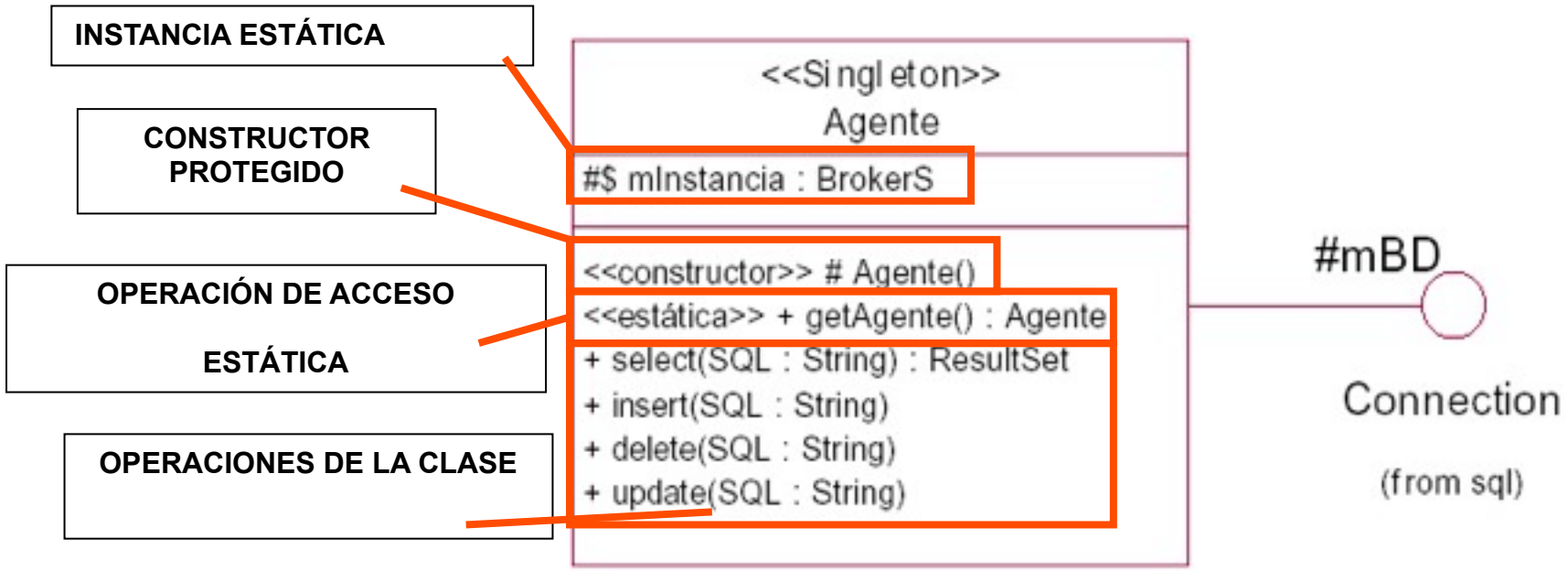
```
public class CuentaJoven extends Cuenta {
 ...
 public void abrir() throws Exception {
    TarjetaDebitoJoven t=new TarjetaDebitoJoven(c);
    RegaloJoven r=new RegaloJoven(r);
 }
}

public class CuentaOro extends Cuenta {
 ...
 public void abrir() throws Exception {
    TarjetaDebitoOro t=new TarjetaDebitoOro(c);
          TarjetaCreditoOro t=new TarjetaCreditoOro(c);
    RegaloOro r=new RegaloOro(r);
 }
}
```

Problema

- Queremos hacer un agente de base de datos (o Broker) en el que se centralice el acceso a la BBDD.
- Queremos asegurarnos de que existe **una única instancia** de ese agente, para que todos los objetos que la usen estén tratando con la misma instancia, accedan a ella de la misma forma,...
- Usar una variable global no garantizaría que sólo se instancie una vez

Singleton

- Los clientes acceden a la única instancia mediante la operación getAgente()
- Esta operación es también responsable de su creación (si existe nos da la instancia y si no existe, la crea)



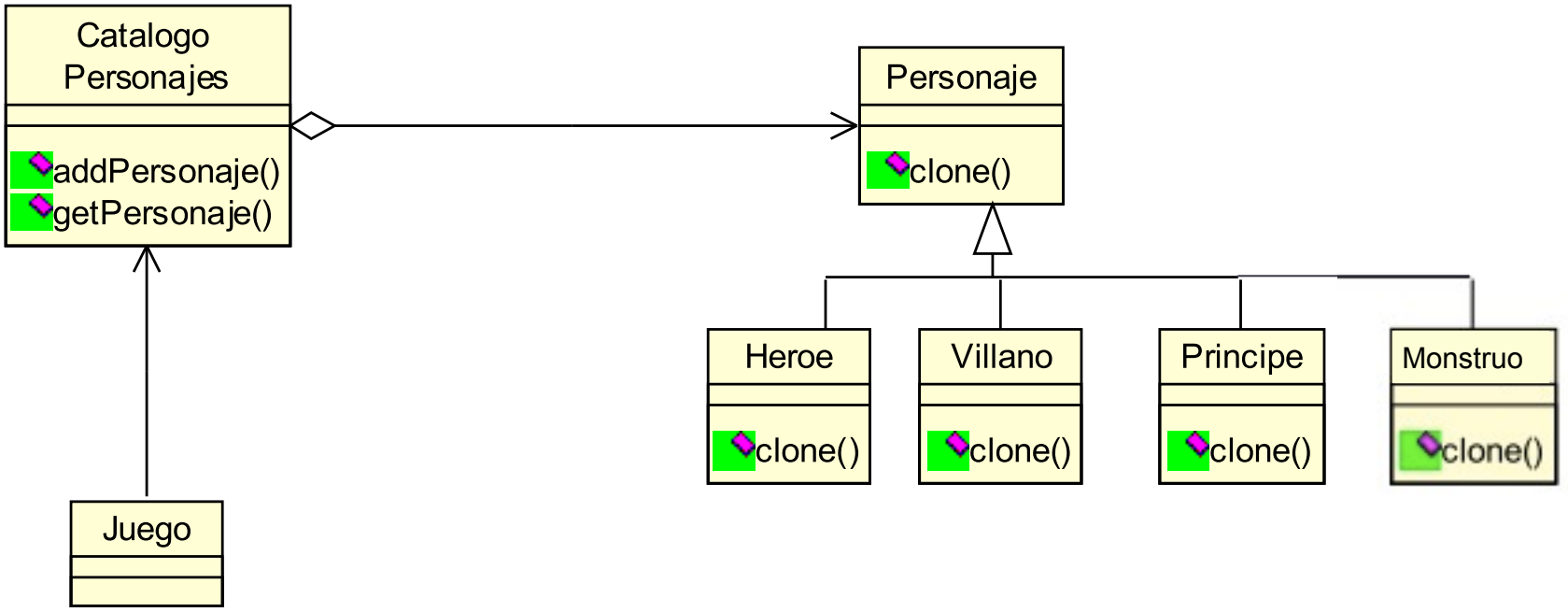
Problema

- Se está desarrollando un juego que permite al usuario interactuar con personajes que juegan ciertos roles.
- Se desea incorporar al juego una facilidad para crear nuevos personajes que se añaden al conjunto de personajes predefinidos.
- En el juego, todos los personajes serán instancias de un pequeño conjunto de clases tales como `Heroe`, `Villano`, `Principe` o `Monstruo`.
- Cada clase tiene una serie de atributos como `nombre`, `imagen`, `altura`, `peso`, `inteligencia`, `habilidades`, etc. y según sus valores, una instancia de la clase representa a un personaje u otro, por ejemplo podemos tener los personajes príncipe bobo o un príncipe listo, o monstruo bueno o monstruo malo.
- Diseña una solución que permita al usuario crear nuevos personajes y seleccionar para cada sesión del juego personajes de una colección de personajes creados.

Prototipo

PROTOTYPE

- Se utiliza el patrón Prototype para mantener un catálogo de instancias prototípicas que se crean a partir de los personajes predefinidos. El catálogo puede implementarse como un singleton.



Problema

- Supongamos que en nuestro sistema bancario...

- La clase “Tarjeta” tiene un método
  - transferir (float importe, String cuentaDestino)
- La clase “Cuenta” tiene un método
  - transferir (float importe, String cuentaDestino)

- El banco ha adquirido un modelo novedoso de cajeros automáticos que permiten hacer transferencias, pero la interfaz esperada para esta operación es:
  - order (String target, float amount)

- ¿ Qué podríamos hacer para que nuestras instancias de Tarjeta y Cuenta respondan adecuadamente a los nuevos cajeros ?

Problema

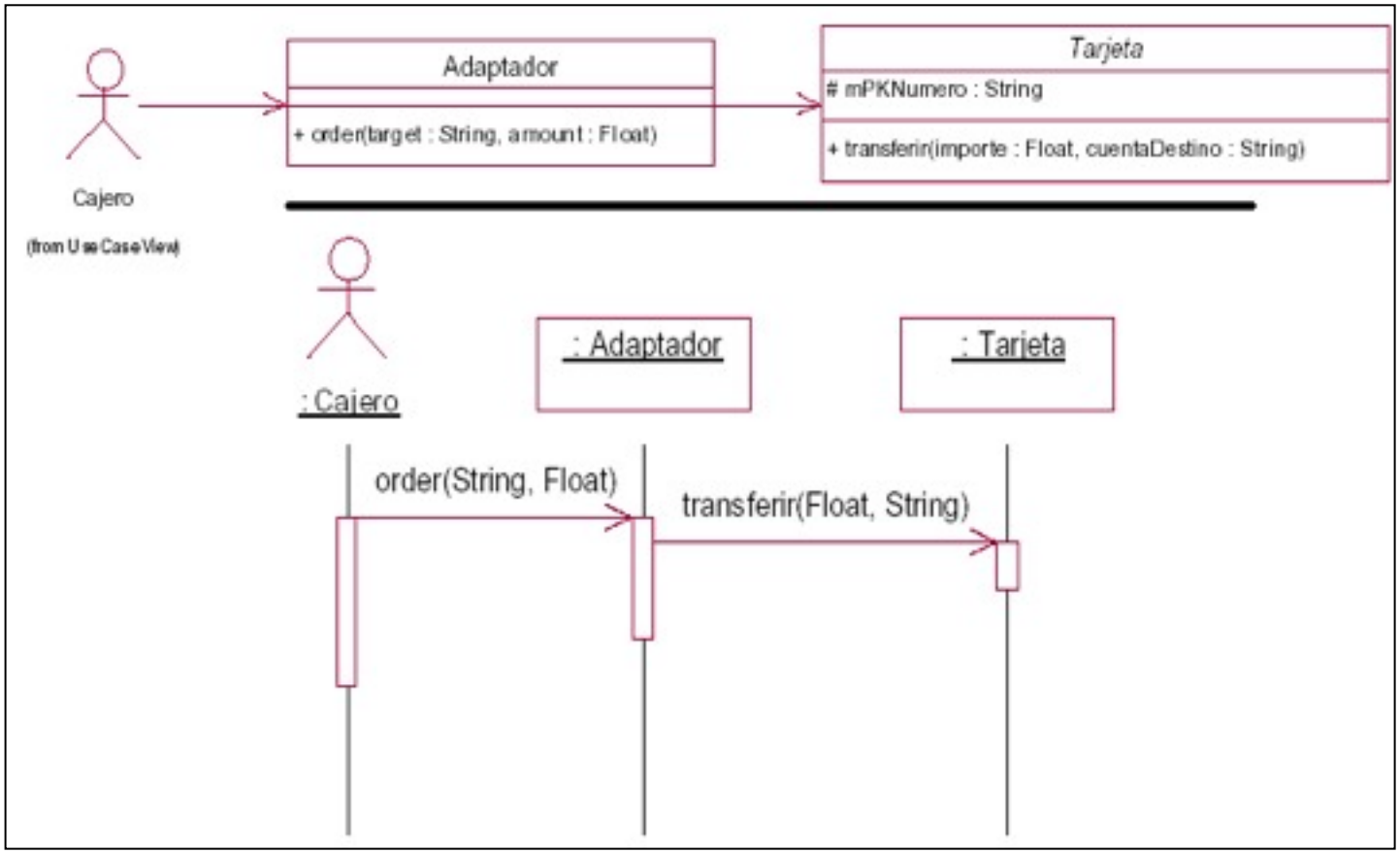
- Solución:
  - Modificar el código de Tarjeta (y de Cuenta), añadiendo un nuevo método
  - ... solución poco elegante

```
// Este método ya existía
 public void transferir(float importe, String cuentaDestino)
         throws Exception {
        if (mCuenta.getSaldo()<importe)
        throw new Exception("Saldo insuficiente");
        if (!mCuenta.getBloqueada())
         throw new Exception("La cuenta está bloqueada");
    mCuenta.transferir(importe, cuentaDestino);
  }

 // Añadimos este otro para que los cajeros puedan comunicar con
 // nuestras tarjetas
 public void order(String target, float amount) throws Exception {
    transferir(amount, target);
 }
}
```

Adaptador

ADAPTER

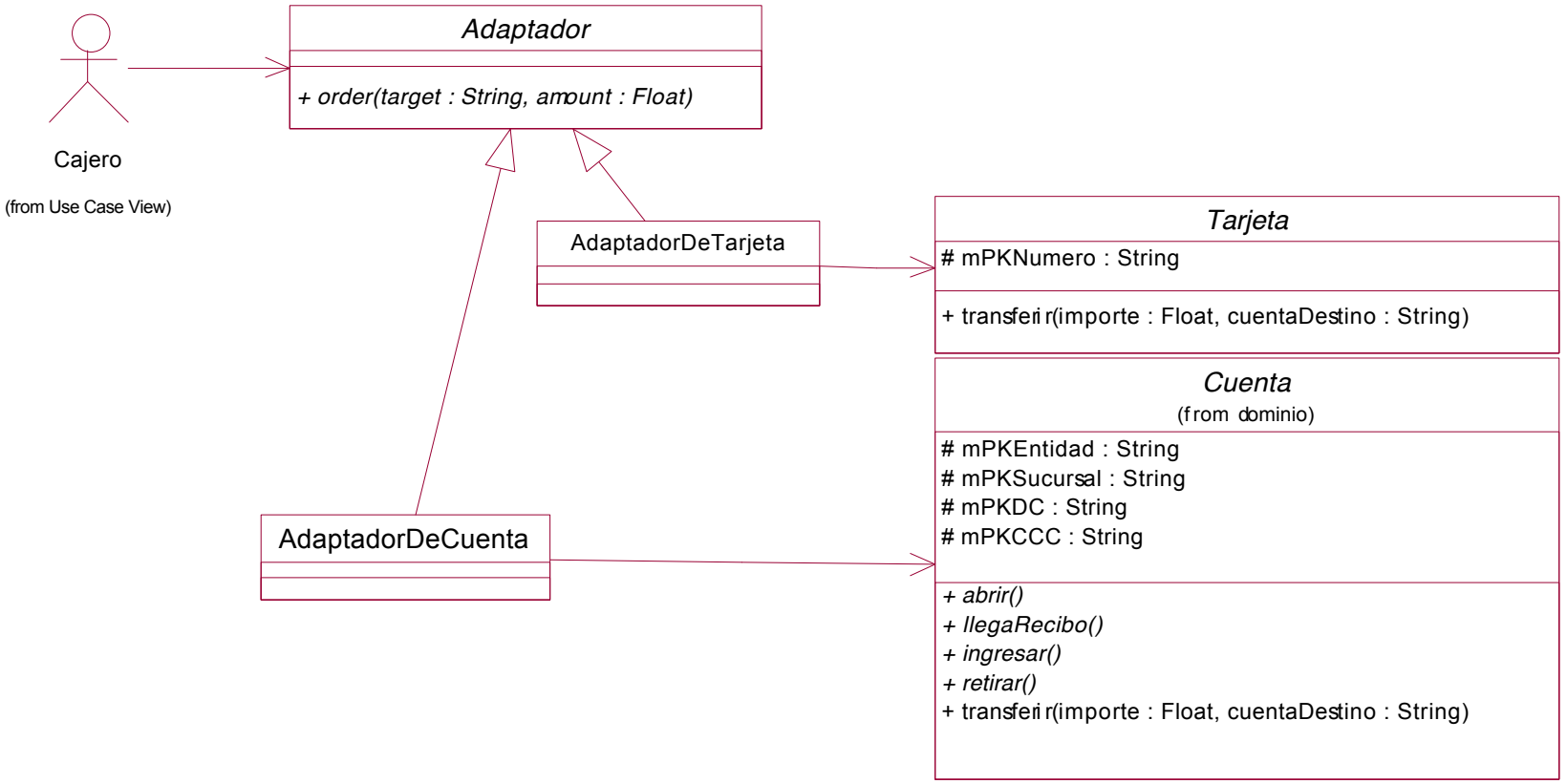
- Solución: Utilizar el patrón Adaptador
- (mucho más elegante y no hay que tocar el código de la clase)



Adaptador

ADAPTER

- El Adaptador podría ser una clase abstracta que sería instaciada según el caso. (hacemos del adaptador una fábrica abstracta).

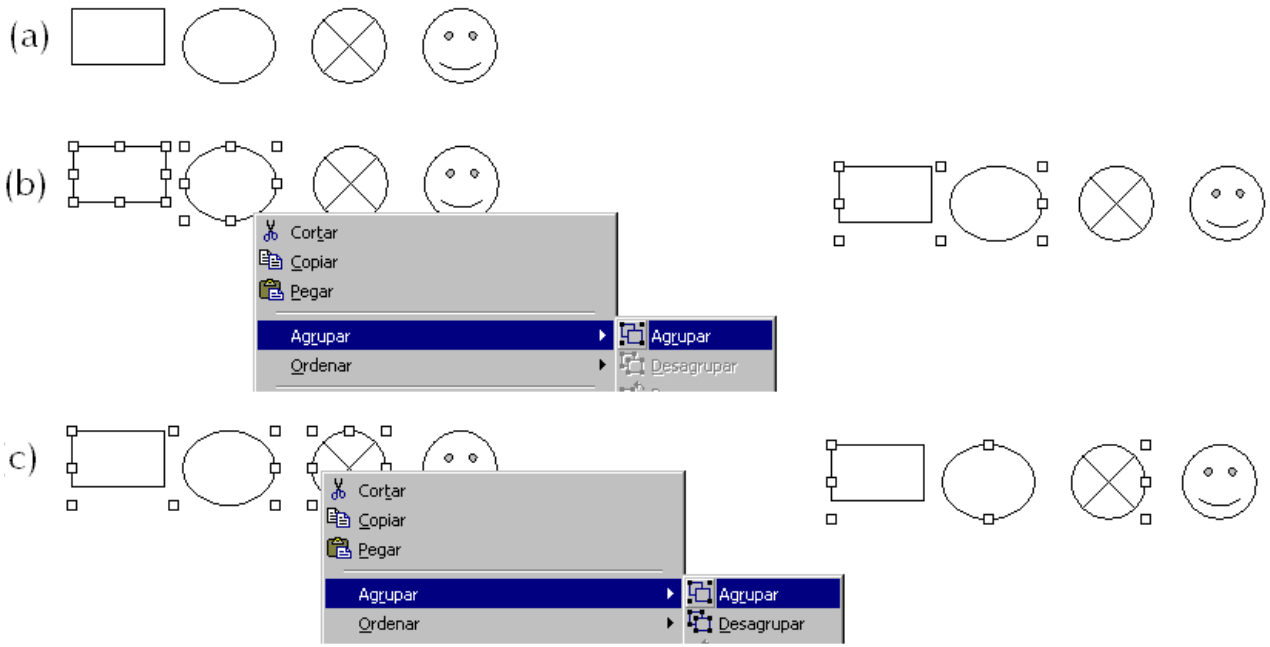


Problema

- Objetos de **dibujo** en un **procesador de texto**.

- Los objetos se pueden **agrupar** en un “grupo” que se maneja como un único objeto.
- Ese “grupo” a su vez se puede agrupar con otros objetos, de forma que cada “grupo” puede contener objetos simples, grupos, mezclados,...

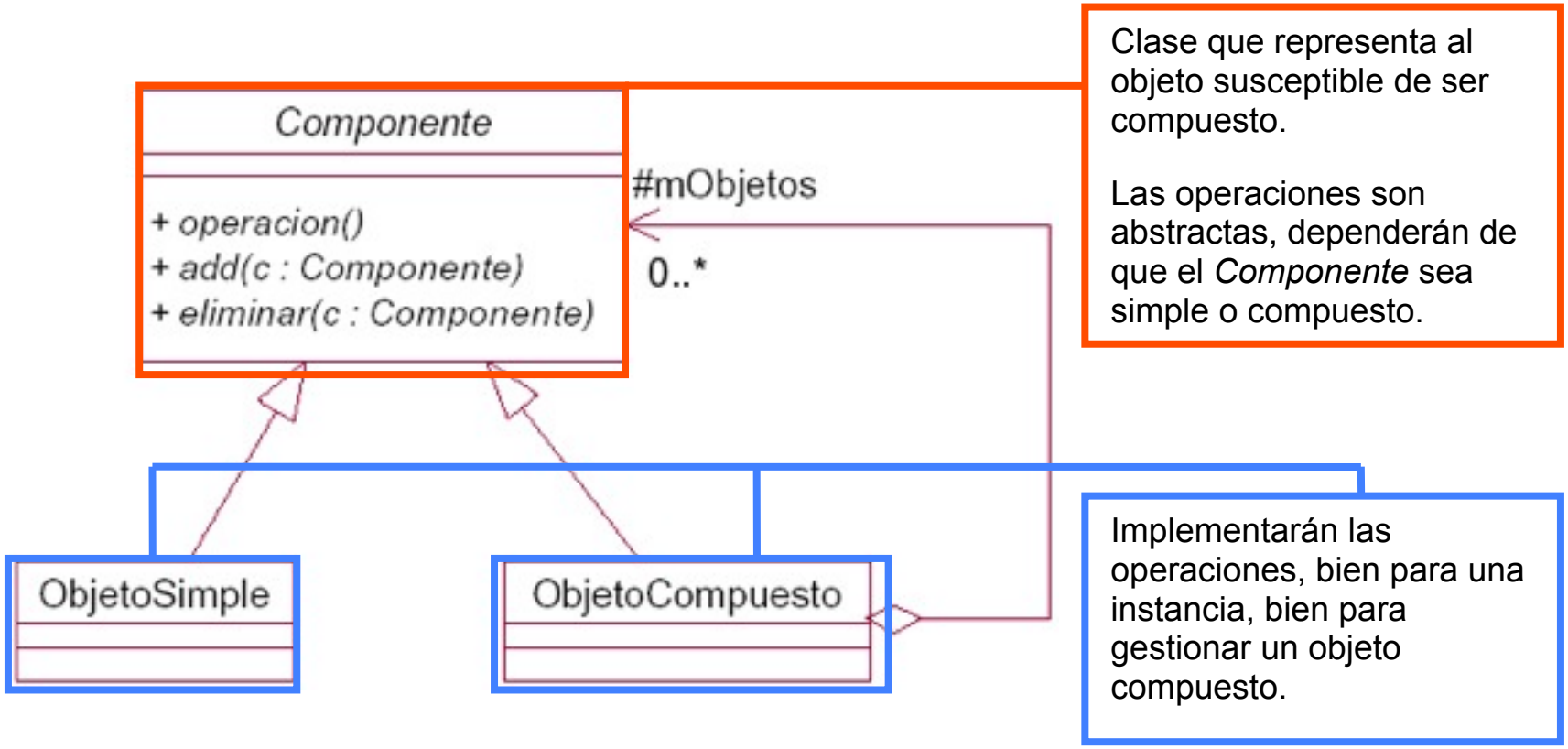
- ¿Cómo
- podríamos
- modelar
- esto?



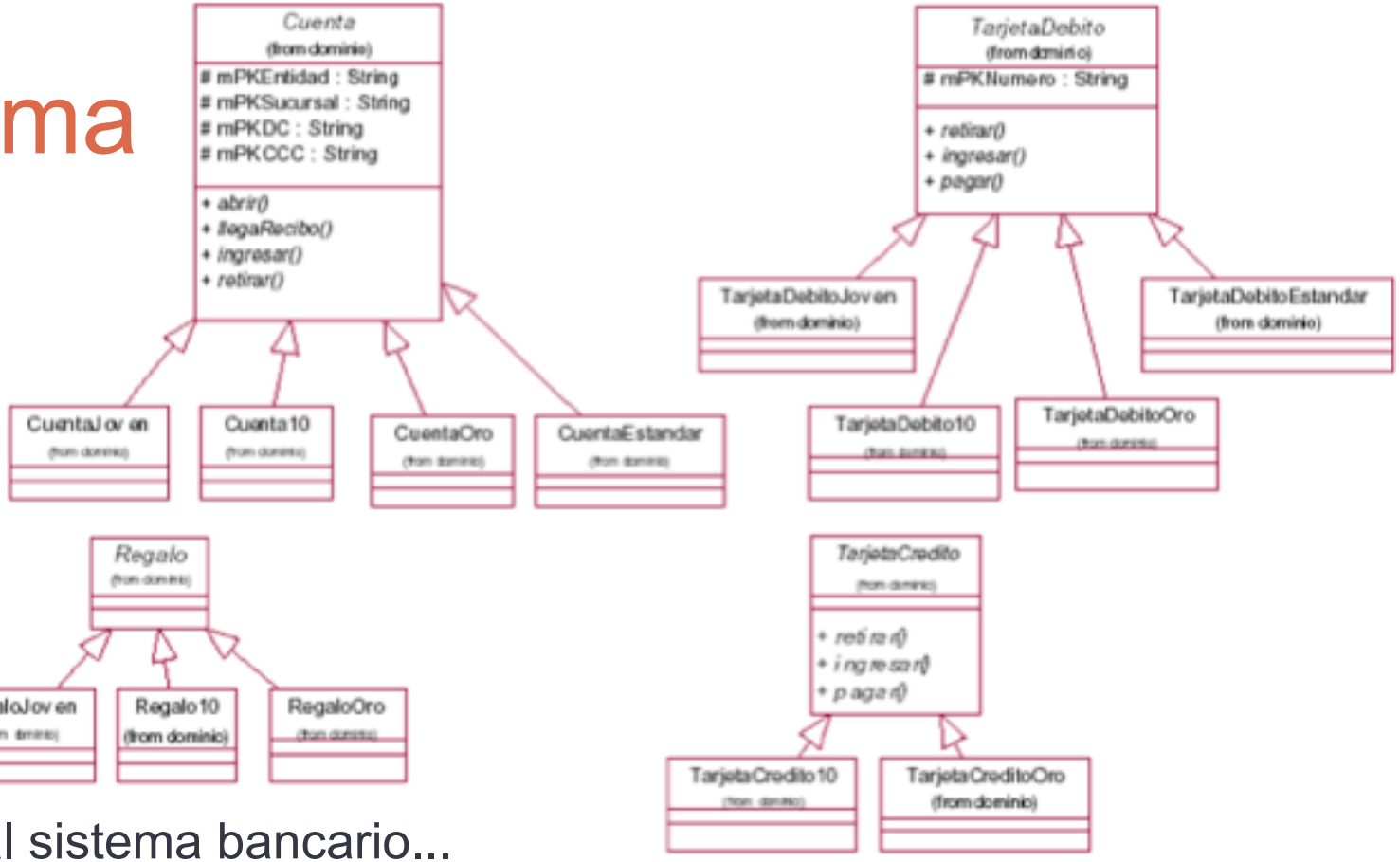
Compuesto

COMPOSITE

- Las dos subclases implementan las operaciones abstractas definidas en “Componente”
- En el ejemplo “ObjetoSimple” sería abstracta y heredarían “rectángulo”, “elipse”,...



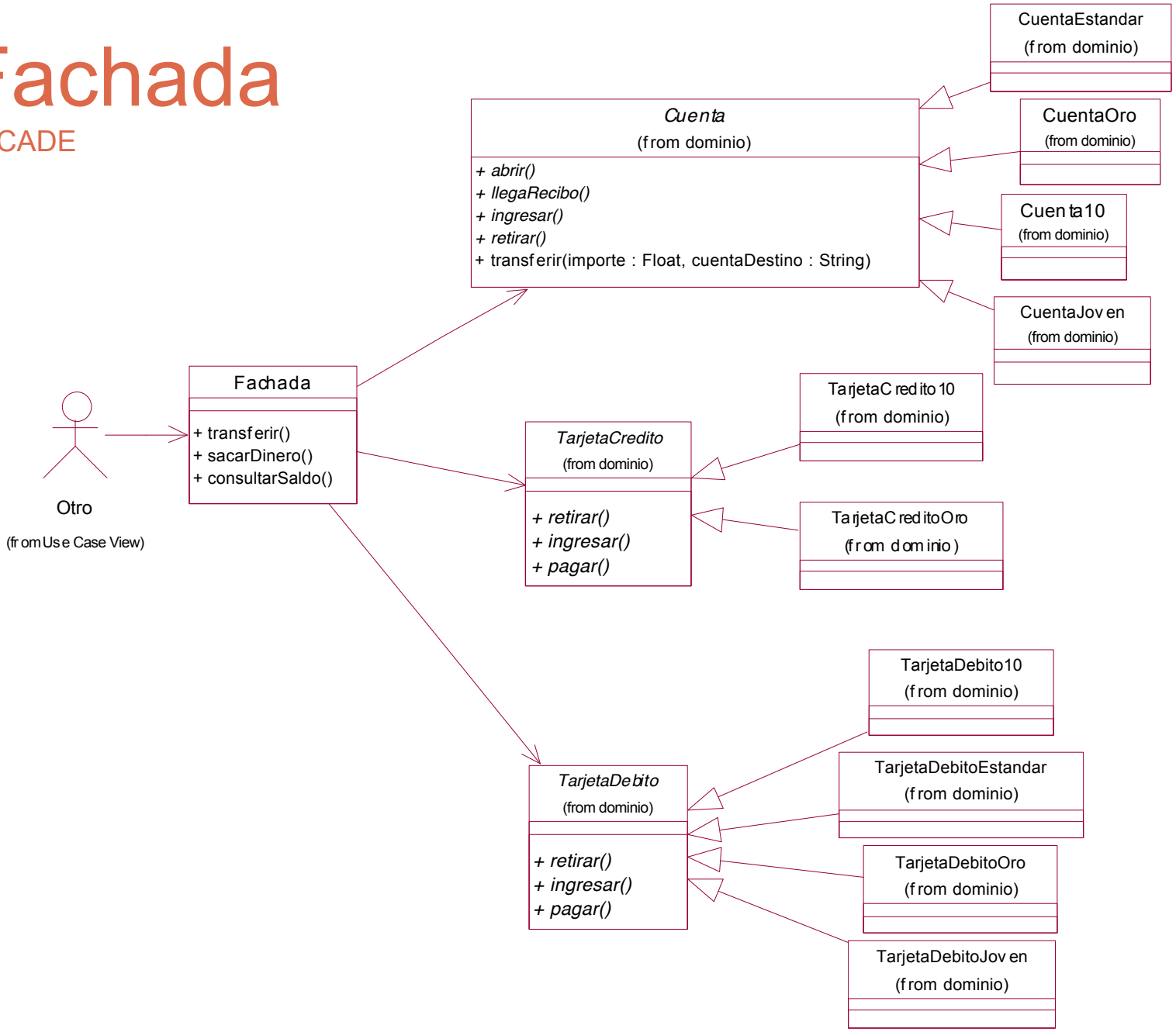
Problema



- Volviendo al sistema bancario...
- Supongamos que deseamos dotar a los desarrolladores de la capa de presentación de un **mecanismo unificado de acceso** a la capa de dominio
- Queremos ofrecer las siguientes operaciones públicas:
  - transferir(), sacarDinero(), y consultarSaldo()
- ¿Cómo podríamos modelar esto?

Fachada

FACADE

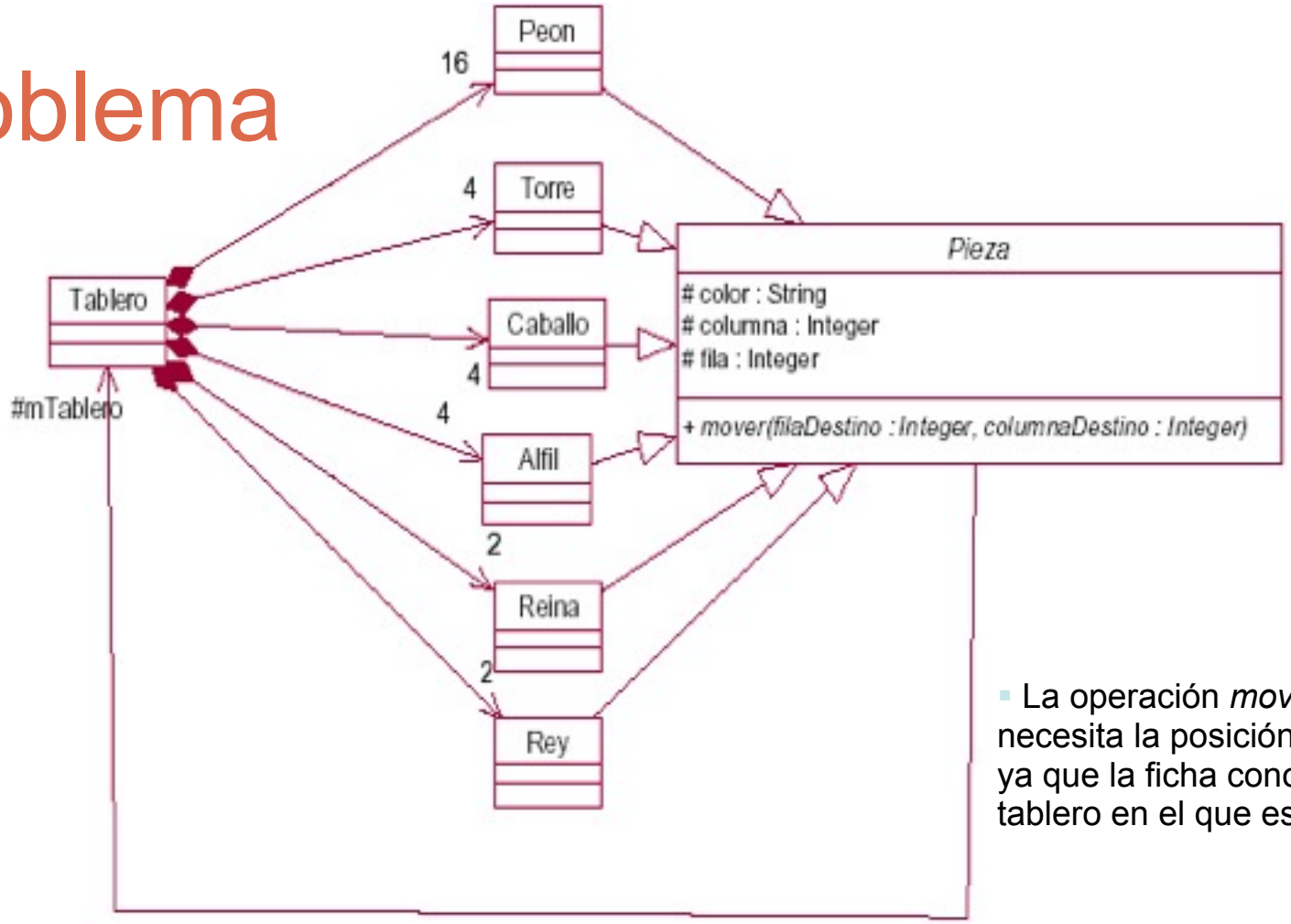


Problema

- Diseño de un tablero de ajedrez
  - Sabemos que hay distintos tipos de piezas
  - Y que cada jugador tiene:
    - 8 peones,
    - 2 torres,
    - 2 caballos,
    - 2 alfiles,
    - 1 reina,
    - 1 rey.

- ¿Cómo podríamos modelar el tablero y las piezas?

Problema

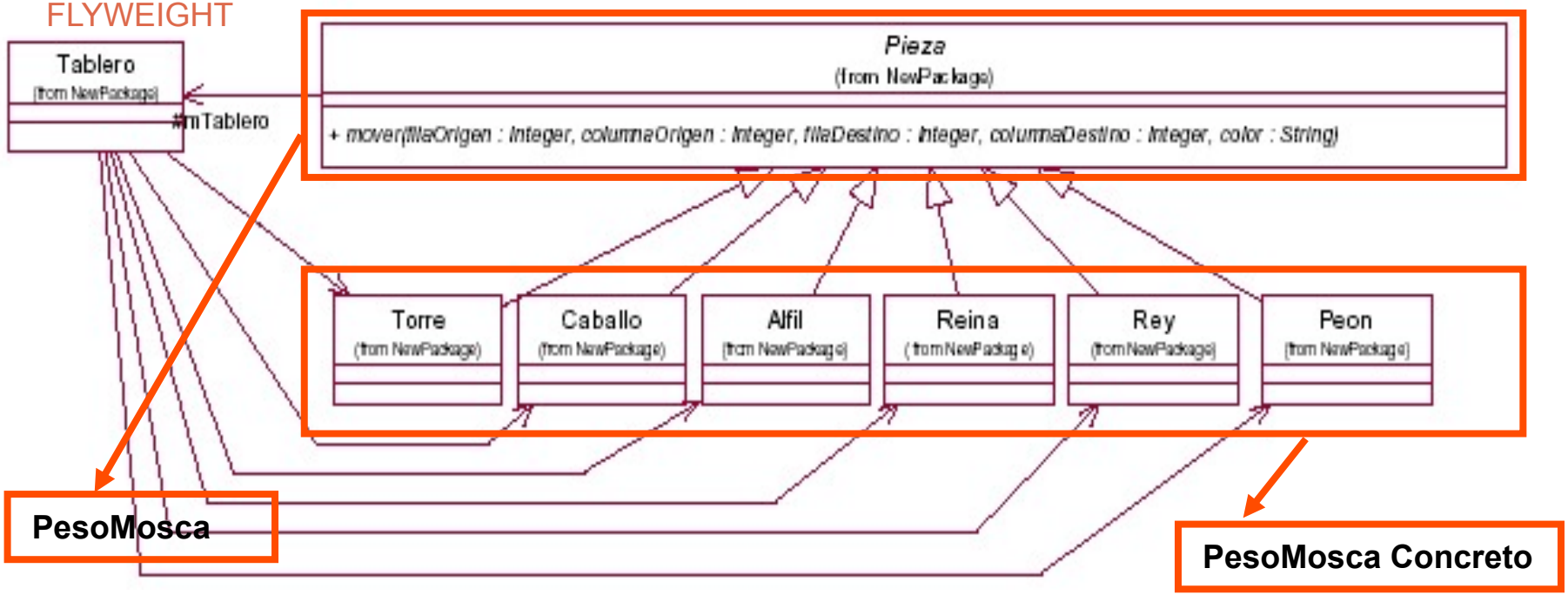


- La operación *mover* tan sólo necesita la posición destino, ya que la ficha conoce al tablero en el que esta ubicada.

- La solución esta bien…pero son 32 instancias por partida.
- Si suponemos que vamos a usar un servidor con múltiples partidas…
- ¿Cómo podríamos modelarlo para disminuir el número de objetos en memoria?

Peso Mosca

FLYWEIGHT



- El tablero conoce sólo a 1 instancia de cada tipo de pieza.
- Pasamos de 32 instancias por partida a 6 (1 por pieza), o incluso a 6 para todas las partidas.
- Estado **intrínseco**: pertenencia a un tipo de pieza y conocimiento del tablero al que pertenece.
- Estado **extrínseco**: ahora a “mover()” le pasamos también el origen y color de la pieza.
- Con esto conseguimos menos memoria pero a cambio de tiempo de cómputo

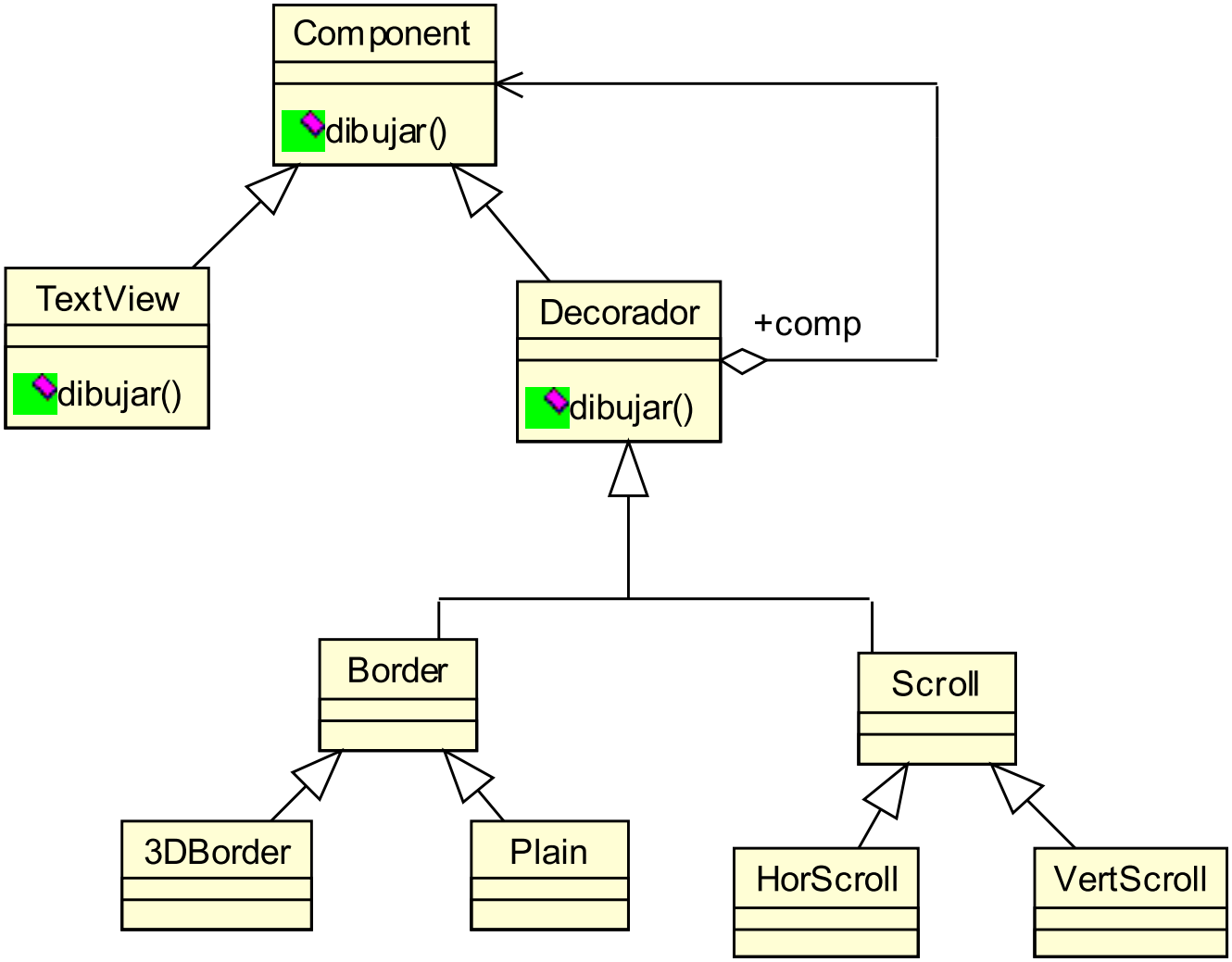
Problema

- Se proporciona una clase `TextView` que representa un componente GUI ventana de texto.
- La clase `TextView` tiene un método dibujar entre otros.
- `TextView` es subclase de `Component` , que es la raíz de la jerarquía de clases que representan componentes GUI.

- Queremos definir ventanas de texto con:
  - diferentes tipos de bordes (*Plain*, *3D*, *Fancy*)
  - y barras de desplazamiento (horizontal, vertical)

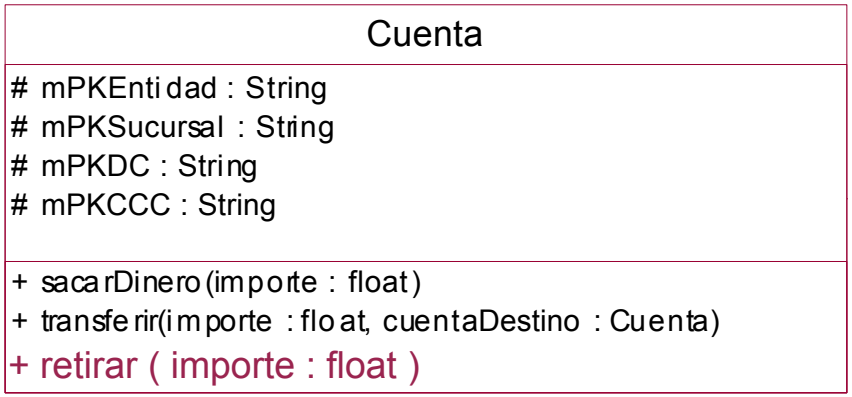
Decorador

DECORATOR



Problema

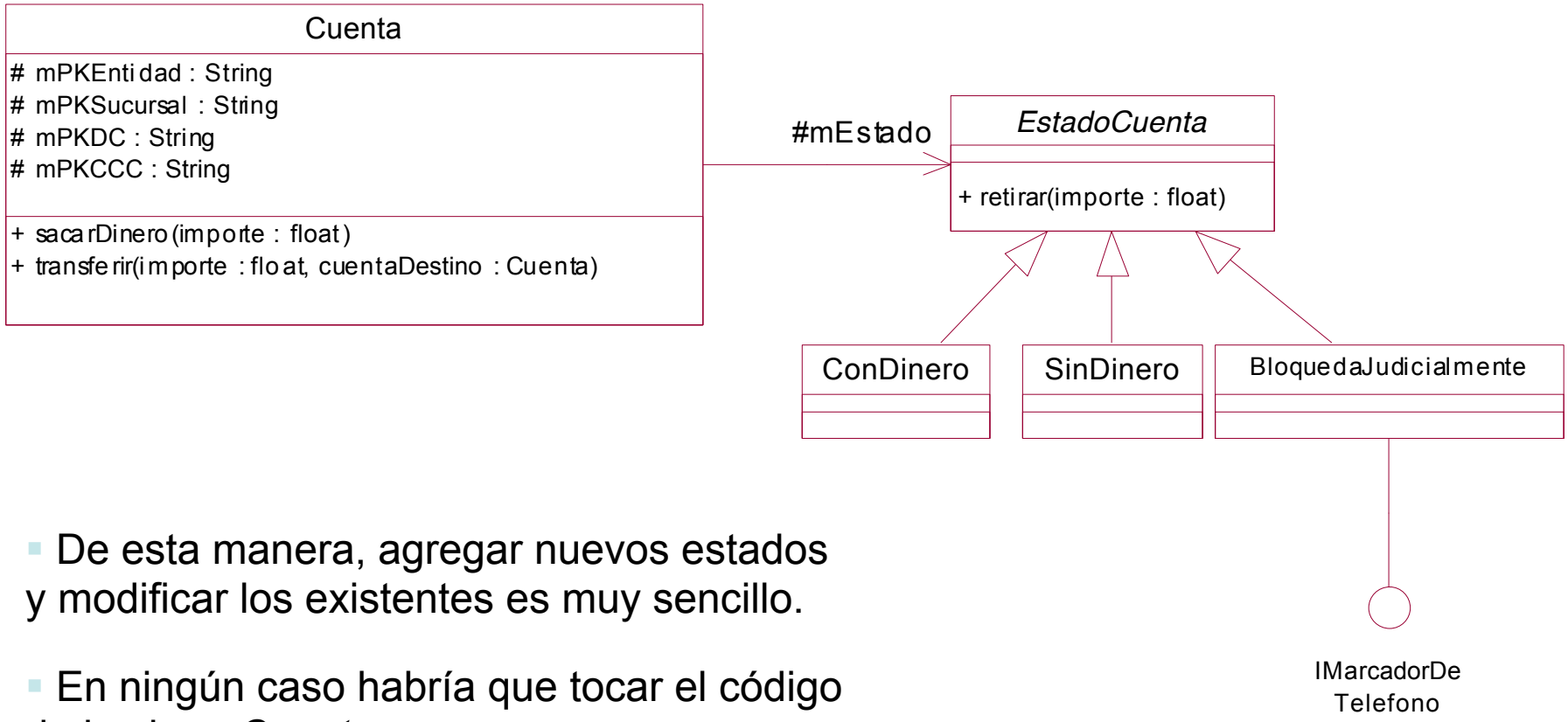
- Sistema bancario
  - Nos centramos en la clase "**Cuenta**"
  - Sabemos que una parte importante del comportamiento de este objeto para la operación "**retirar**(**float importe**)" depende del estado de la cuenta, que puede ser:
    - Cuenta ConDinero
    - Cuenta SinDinero
    - Cuenta BloqueadaJudicialmente



  - Queremos delegar el comportamiento de la operación **"retirar"**, ¿cómo lo haríamos?

Estado

STATE



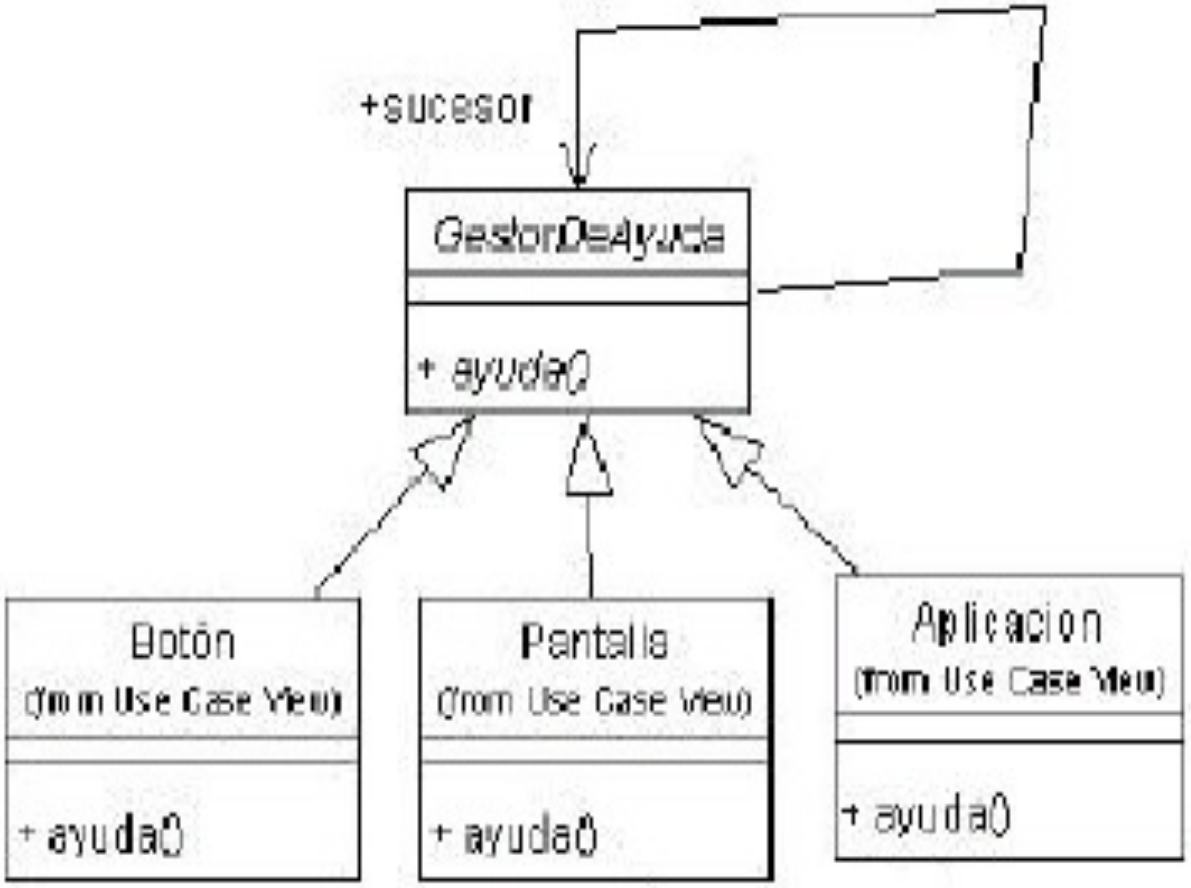
- De esta manera, agregar nuevos estados y modificar los existentes es muy sencillo.
- En ningún caso habría que tocar el código de la clase *Cuenta.*

Problema

- Gestor de ayuda
  - Tenemos varias clases “Boton”, “Pantalla” y “Aplicacion”.
  - Cada una con un método “ayuda()”.
  - Queremos desacoplar al emisor de un mensaje del receptor, de forma que se pueda ejecutar la operación “ayuda()” en cualquier objeto.
  - Si el objeto es capaz de hacerse cargo de la operación, la ejecutará, y si no la pasará a otro (sucesor) hasta que alguien se haga cargo del mensaje.
- ¿Cómo podríamos modelarlo?

Cadena de Responsabilidad

CHAIN OF RESPONSIBILITY



Problema

- Procesamiento de un lenguaje sencillo
  - Supongamos que en el sistema bancario es posible realizar algunas operaciones sobre las cuentas con un lenguaje de comandos sencillo, que viene dado por la siguiente gramática:

```
operación     : retirada | ingreso | transferencia
retirada      : 'retirar' CANTIDAD CUENTA
ingreso       : 'ingresar' CANTIDAD CUENTA
transferencia : 'transferir' CANTIDAD CUENTA CUENTA CONCEPTO
```

- ¿Cómo podríamos modelarlo?

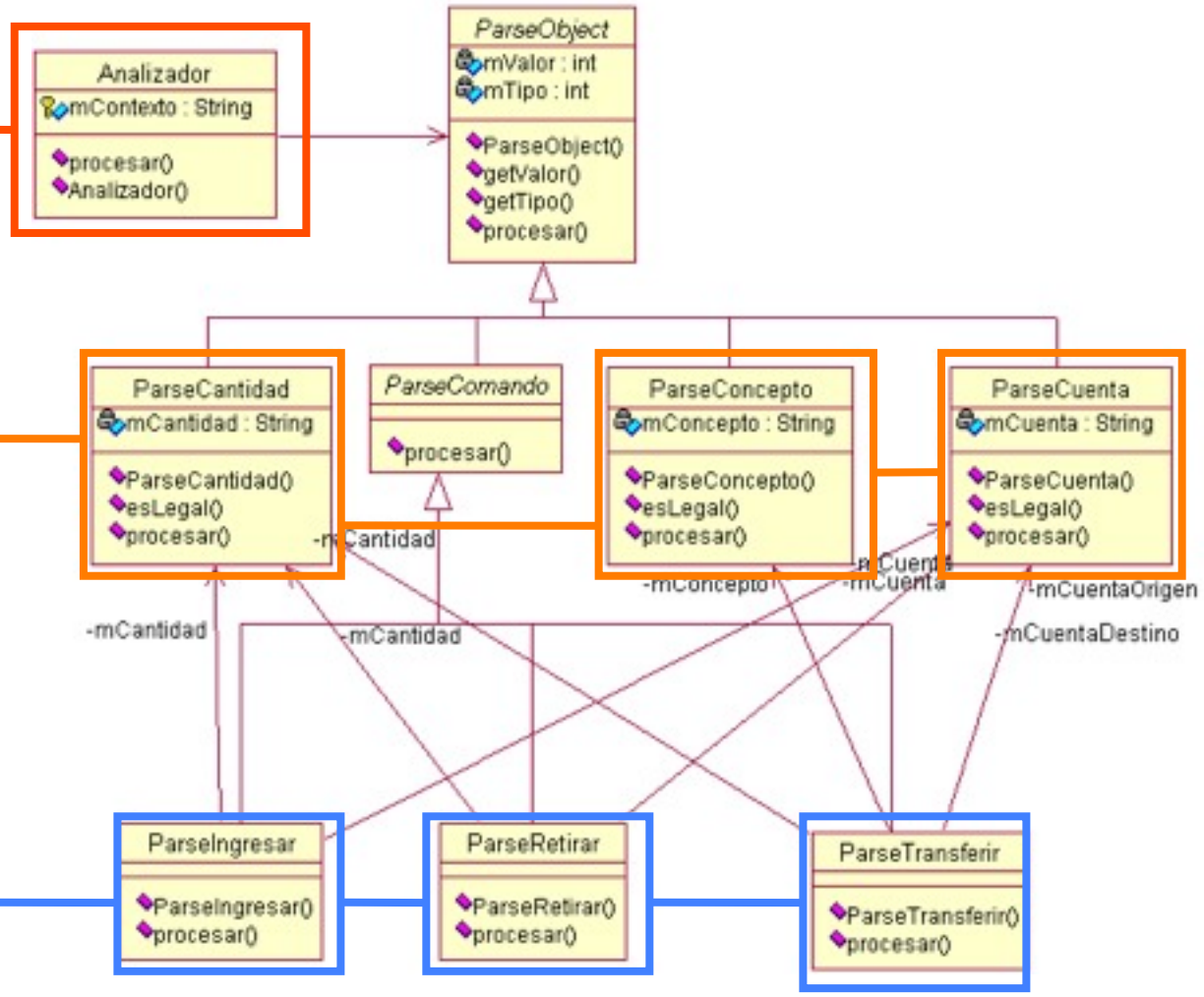
Intérprete

INTERPRETER

Representa el árbol sintáctico de la expresión a analizar.

Los terminales se convierten también en clases

Una clase por cada una de las reglas

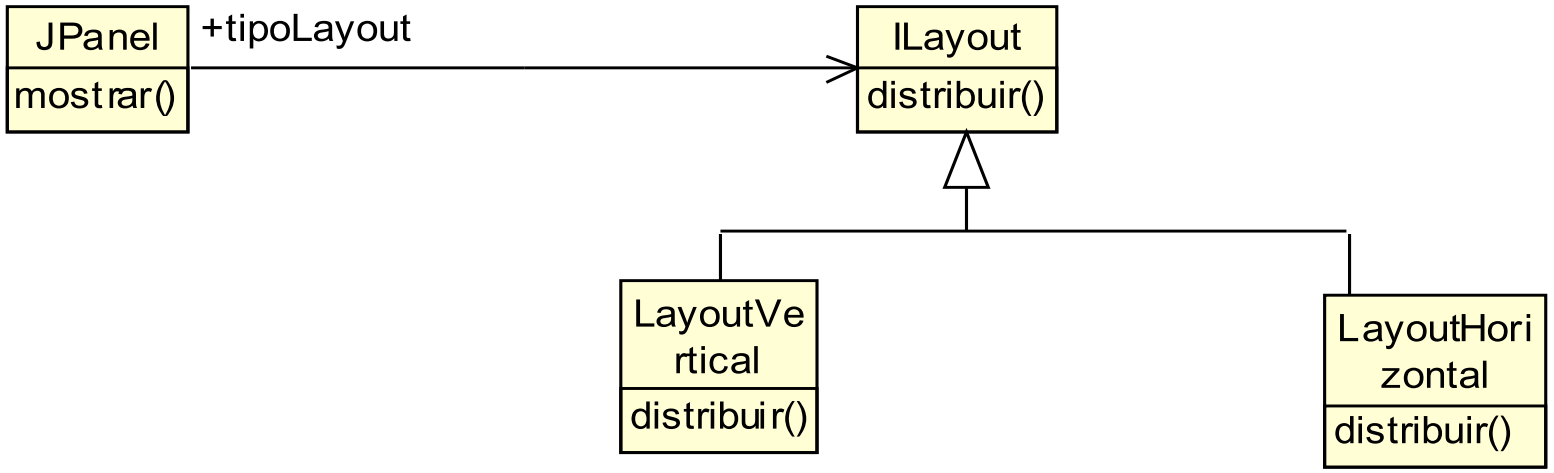


Problema

- Se está construyendo una librería de clases para representar componentes GUI.
- Se ha decidido que en vez de que un programador defina la posición de los componentes GUI (*Button*, *List*, *Dialog*,..) sobre una ventana, se incluyan manejadores de disposición de componentes (*layout manager*).
- Cada uno distribuye un conjunto dado de componentes gráficos de acuerdo a algún esquema de distribución: horizontalmente, verticalmente, en varias filas, en forma de una matriz, etc.
- Debe ser posible cambiar en tiempo de ejecución la distribución elegida inicialmente.
- Supuesto que la clase `JPanel` es la que representa a un contenedor de componentes gráficos, diseña una solución para introducir en la librería los manejadores de disposición.
- Dibuja el diagrama de clases que refleje la solución.

Estrategia

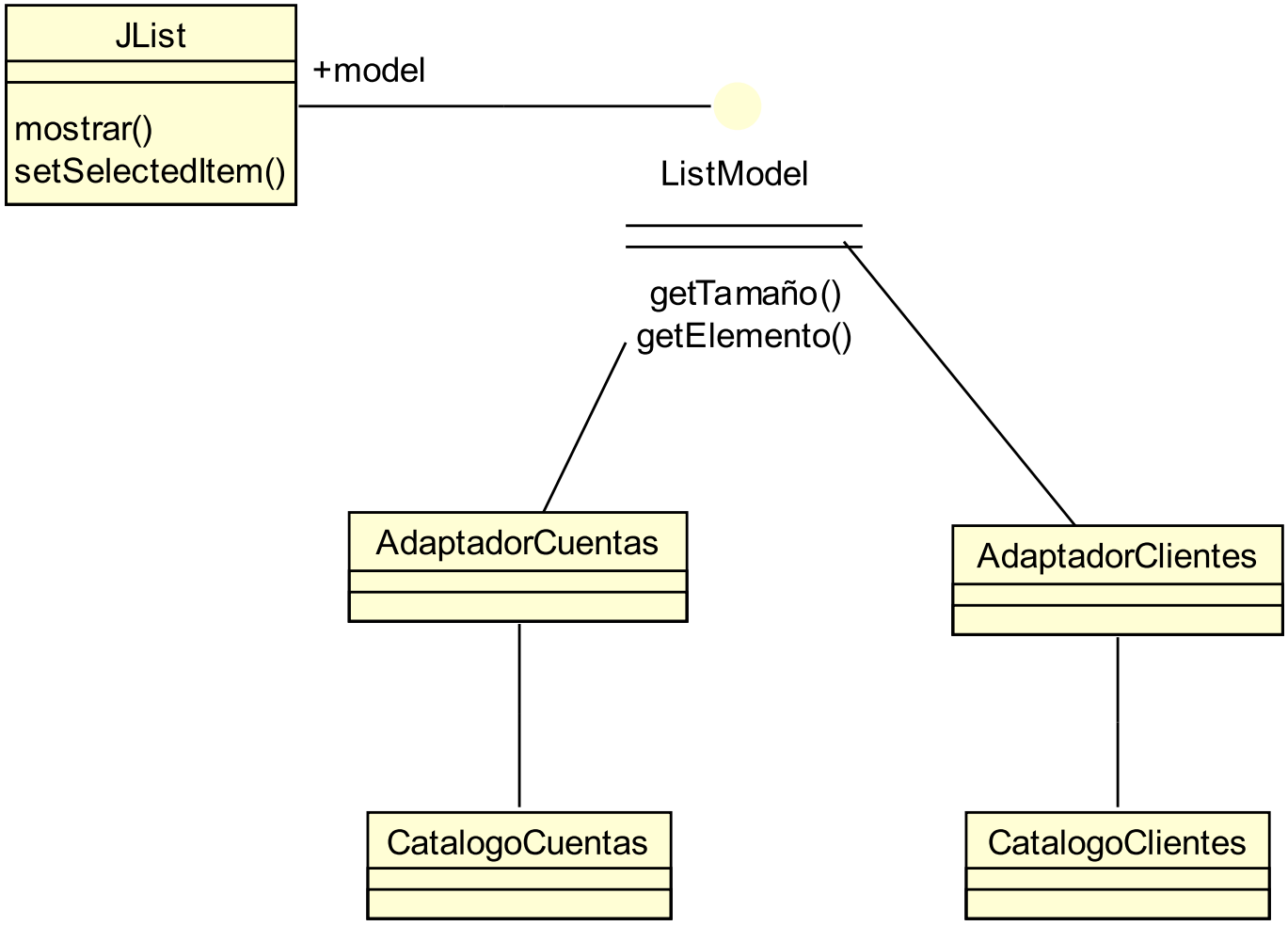
STRATEGY



Problema

- La librería Swing de Java incluye la clase `JList` para presentar una lista de objetos (el texto que se visualiza es determinado por el método `toString` de la clase de los objetos) y permitir al usuario seleccionar uno de ellos.
- El diseño de esta clase es un ejemplo de utilización del patrón `Adaptador` para conseguir una clase reutilizable; se consigue que `JList` sea independiente de la fuente (o modelo) de datos.
- Muestra mediante un diagrama de clases el diseño que permitiría a una clase como `JList` visualizar diferentes listas de items, como por ejemplo un catálogo de clientes (por ejemplo, se mostraría su NIF) o un catálogo de cuentas (por ejemplo, se mostraría el código de cuenta).
- Para cada clase o interfaz del diagrama indica los métodos y atributos que son significativos.
- Nótese que la clase `JList` necesita disponer de información como el número de ítems a visualizar y el objeto seleccionado.
- Los adaptadores almacenan la lista que se visualiza como una instancia de `Vector`.

Solución


JList
mostrar()
setSelectedItem()
+model
ListModel
getTamaño()
getElemento()
AdaptadorCuentas
AdaptadorClientes
CatalogoCuentas
CatalogoClientes