

fiuba

algo3

# **Temas de diseño**

Carlos Fontela

[cfontela@fi.uba.ar](mailto:cfontela@fi.uba.ar)

# Temas de diseño



# ¿Diseño?

Concebir una solución para un problema

Es el “cómo” del desarrollo

Fase más ingenieril del desarrollo de software (CF)

Entre el “qué” de las necesidades del usuario y la propia construcción del producto



# Temario

Diseño y patrones

Patrones macro

MVC como caso particular

Modularización

Principios de diseño

Para clases

Para paquetes

Para métodos

# Patrones

Solución exitosa de un problema que se presenta con frecuencia

“Alguien ya ha resuelto nuestros problemas”

Patrones de programación

Algoritmos “con nombre”

Métodos de ordenamiento, búsqueda

Incluyen el algoritmo y una estructura de datos

Patrones de diseño

Para problemas de diseño

# Uso de patrones

Aplicación en distintos contextos

Comunicación con un vocabulario común

Y enseñanza

Comprensión de un sistema ya construido

Hay en las demás ingenierías

Y en otras disciplinas



# Diseño y patrones



fiuba  
algo3

# Qué diseñamos

Despliegue en hardware

Subsistemas o componentes de software y sus interacciones

Integración con otros sistemas

Interfaz de usuario

Estructuras de datos

Algoritmos

Definiciones tecnológicas: plataforma, lenguaje, base de datos, etc.



# Diseño de IU

O Diseño externo

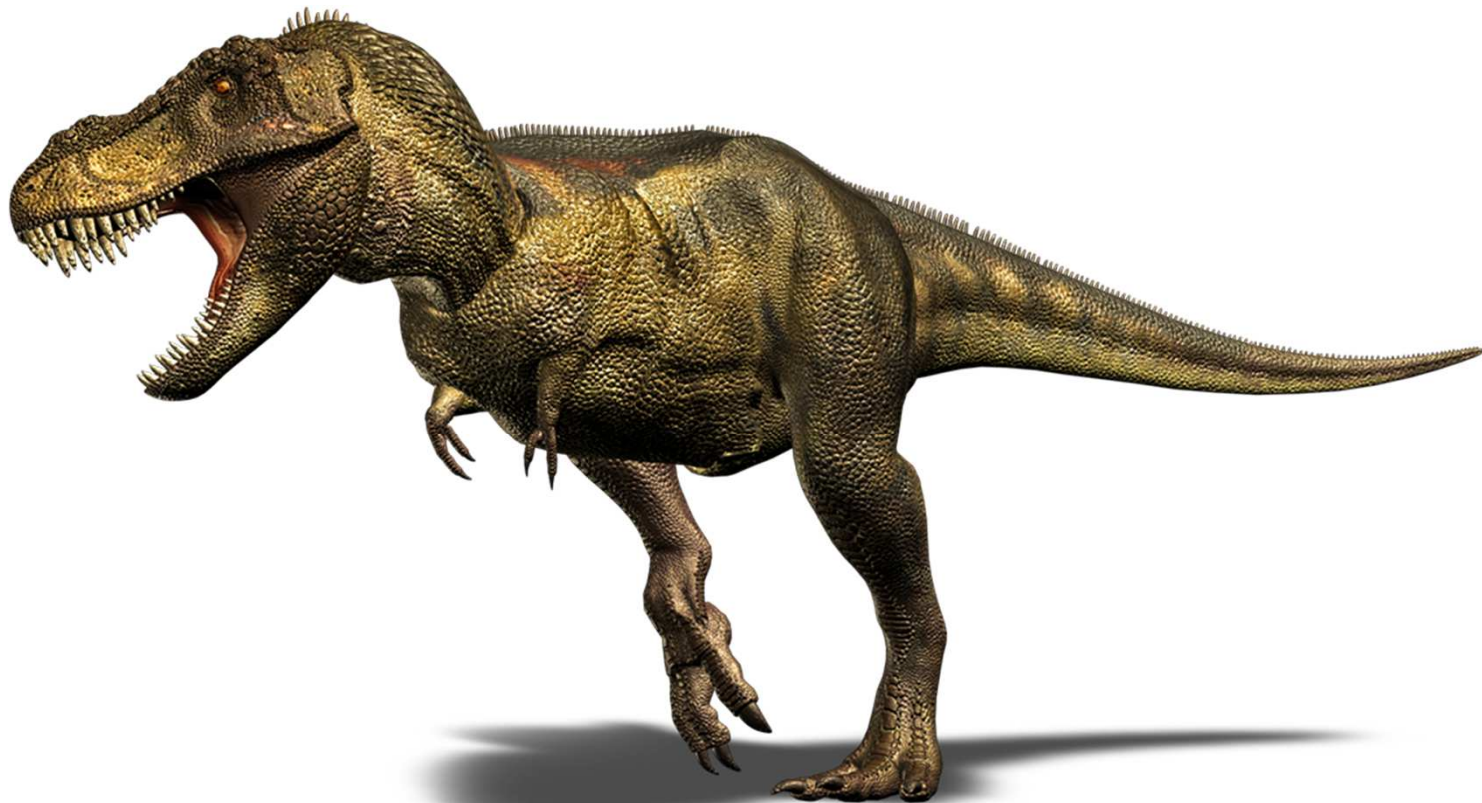
Usabilidad, UX y afines



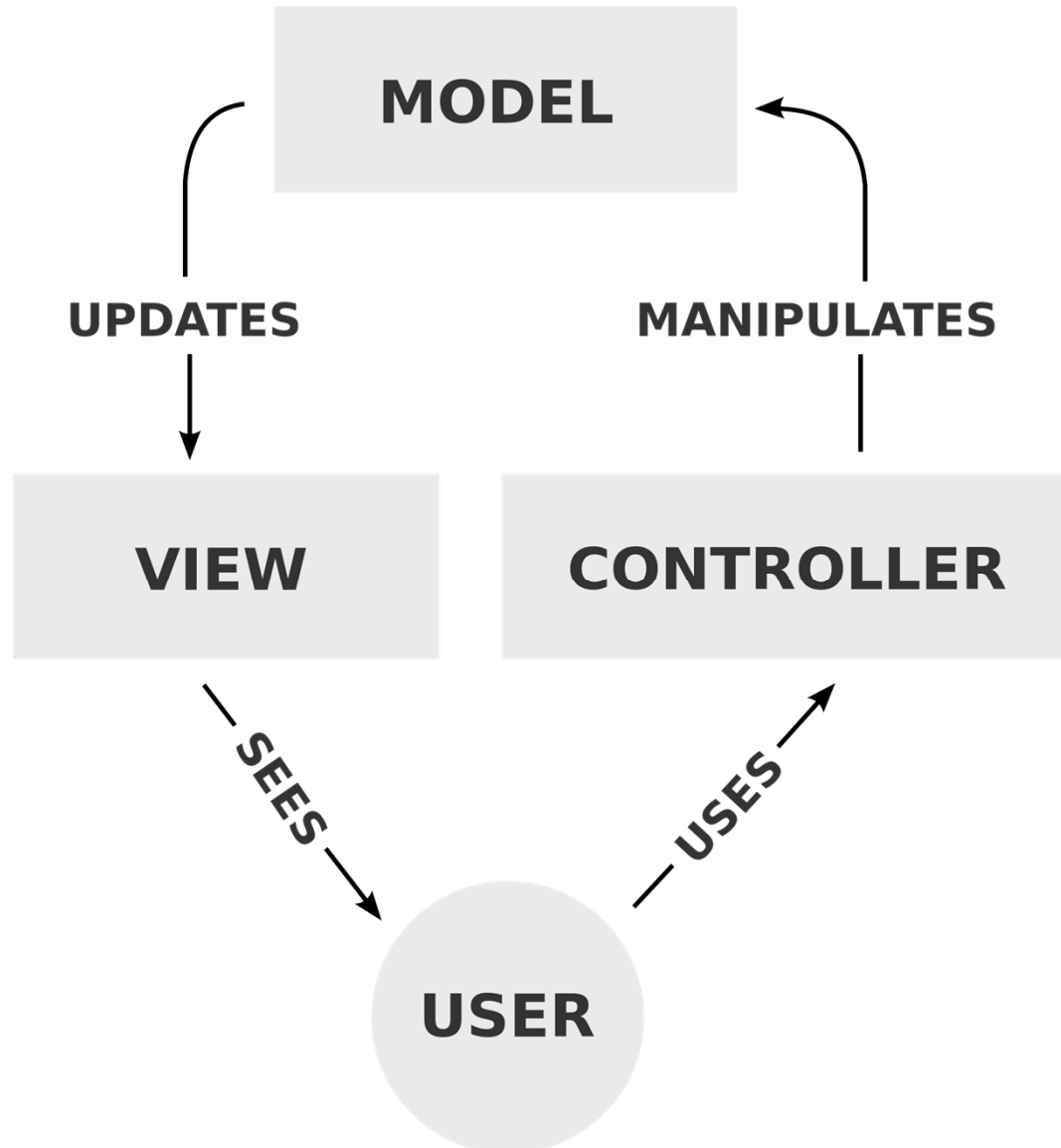
Ver diapositivas en el campus

# Principio Nº 1

¡Prepararse para el cambio!  
Prever el cambio



# MVC: un patrón macro



# MVC: ventajas

Separa incumbencias

Facilita cambios

Bueno en aplicaciones de mucha IU

Muchas vistas por modelo

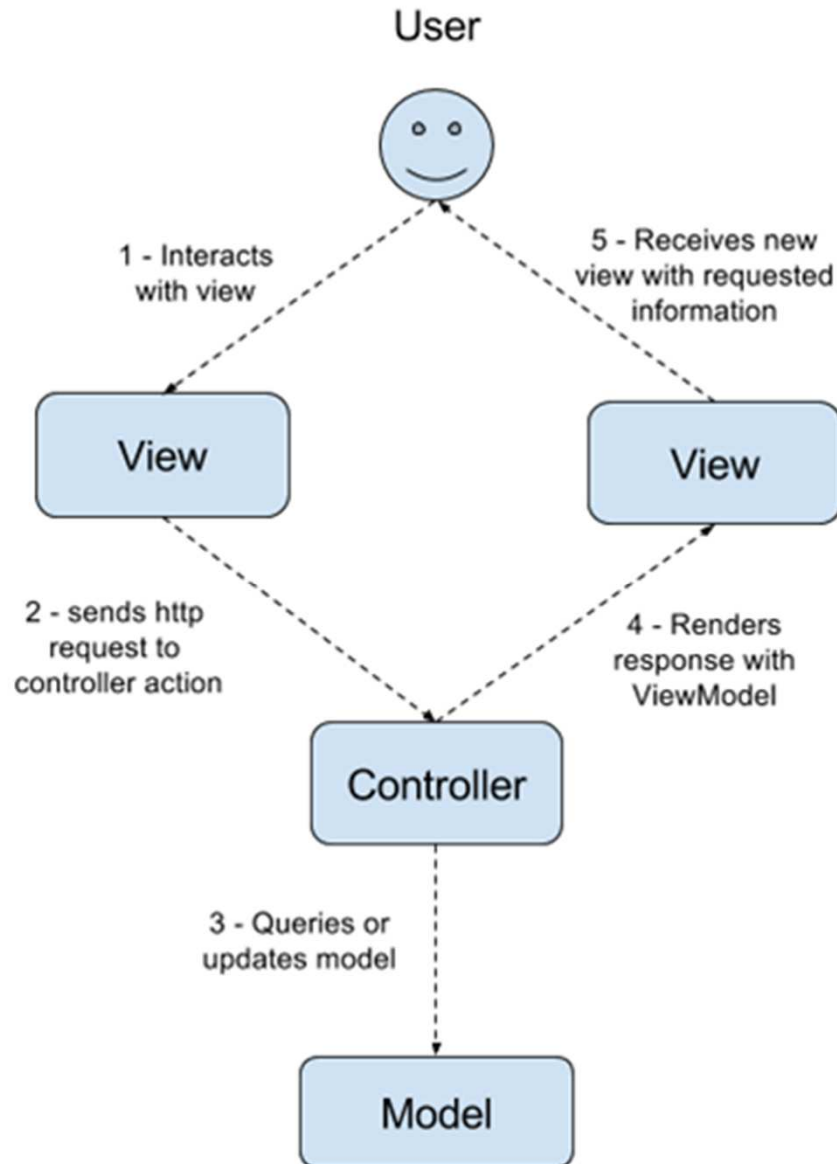
¿Y la persistencia?

En el Modelo

MVC-P



# MVC Web

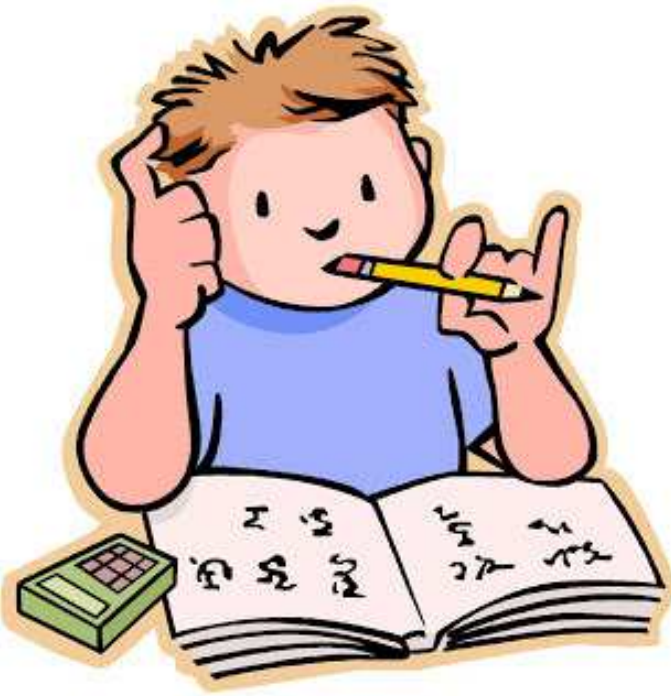


# Recapitulación



# Recapitulación: preguntas

- ¿A qué llamamos diseño?
- ¿Para qué sirven los patrones?
- ¿Qué problemas resuelve MVC?





# Modularización



fiuba  
algo3



# Cohesión y acoplamiento

Para mantener comprensibles, mantenibles y reutilizables a los módulos

## Módulos de alta cohesión

Se refiere a la coherencia intrínseca del módulo

Visión interna y externa

Pero respecto del módulo solamente

## Módulos de bajo acoplamiento

Se refiere a la relación con otros módulos

Visión externa

Mira más bien al sistema

# Cohesión y acoplamiento en clases

La clase representa una sola entidad (un sustantivo debe poder describirla)

Para lograr alta cohesión

Integridad conceptual

La clase debe tener pocas dependencias con otras clases

Para lograr bajo acoplamiento

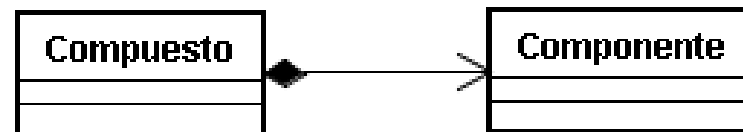
Modificaciones tienen efectos acotados

# Niveles de dependencia entre clases

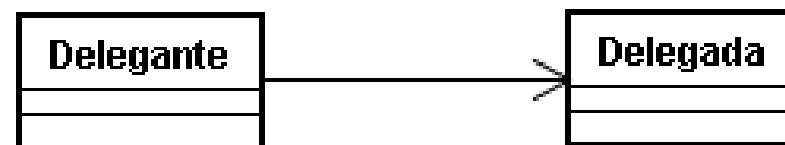
Herencia (de clases o interfaces)



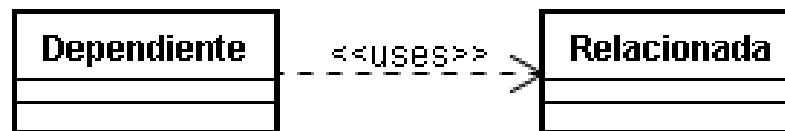
Composición



Asociación simple



Dependencia débil



# Cohesión y acoplamiento en paquetes

Los paquetes deben tener pocas dependencias de otros paquetes

Dependencia de paquetes: cuando las clases de un paquete dependen de la de otro

Para lograr bajo acoplamiento

Modificaciones tienen efectos acotados

Los paquetes deben describir una visión global del sistema

Con clases relacionadas entre sí

Para lograr alta cohesión

# Cohesión y acoplamiento en métodos

Cada método debe poder describirse con una oración que tenga un solo verbo activo

Para lograr alta cohesión

Mejora la legibilidad de todo el código

El método debe tener pocos parámetros y pocas llamadas a otros métodos

Para lograr bajo acoplamiento

Modificaciones tienen efectos acotados

Parámetros por valor en lo posible

# Recapitulación

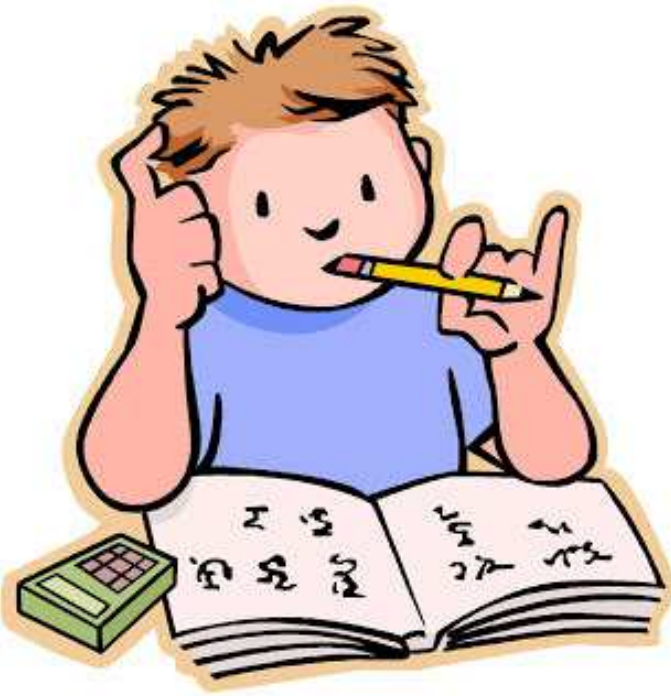


# Recapitulación: preguntas

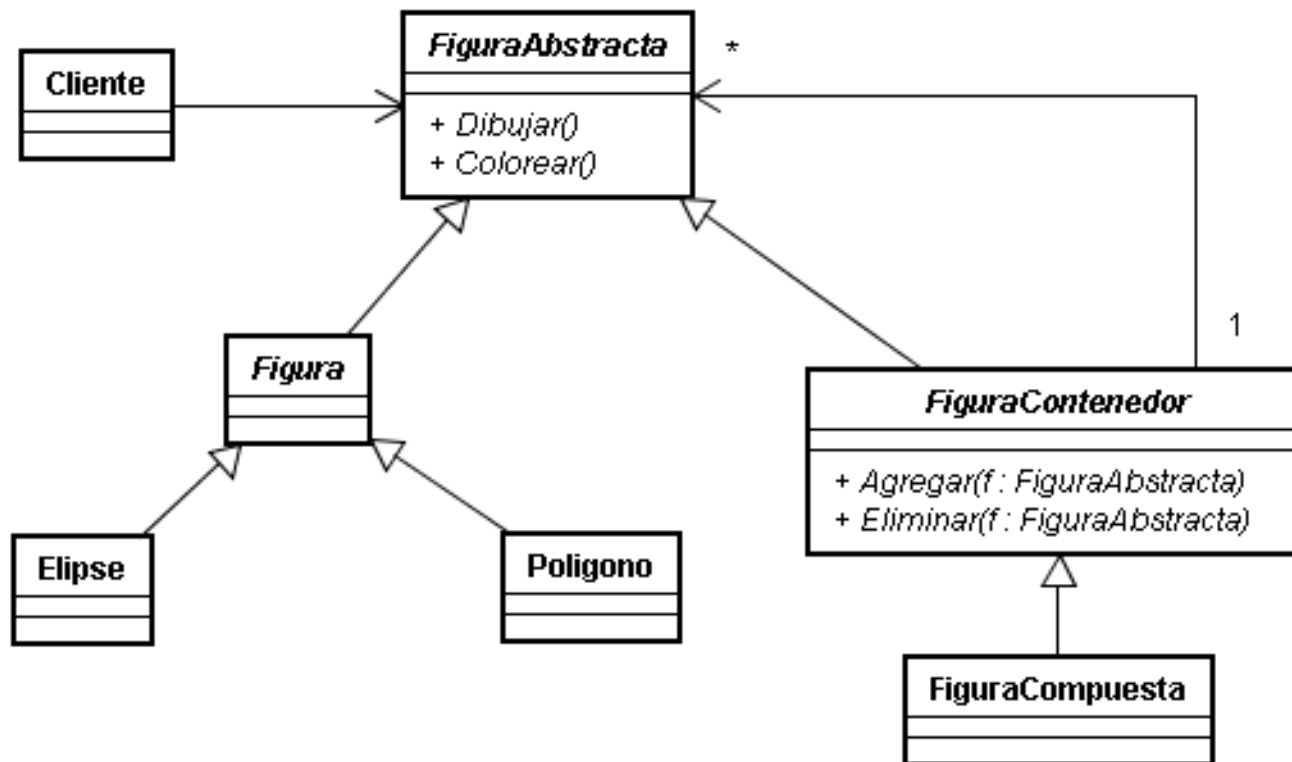
¿Qué es el acoplamiento?

¿Qué es la cohesión?

Analizar MVC a la luz de estos conceptos



# Diseño de clases





# Razones para crear clases (1)

Modelar una entidad del dominio

Modelar una entidad abstracta

- Al encontrar elementos en común entre varias clases

- Clase abstracta o interfaz, cuando nos interese mantener la interfaz

- Clase delegada, cuando nos interesen los datos en común

Reducir complejidad

- Una sola abstracción por clase

Agrupar operaciones relacionadas en una clase de servicios

- Como la clase *Number* de Smalltalk

## Razones para crear clases (2)

Aislar complejidad y detalles de implementación en una clase aparte

Velocidad	y no	Vector3D
Dinero	y no	double
ListaEmpleados	y no	ArrayList<Empleado>

Aislar una porción de código que se espera que cambie respecto de otras más estables

Al separar el código poco claro en clases especiales

Una nueva clase descendiente debe añadir o redefinir un método (modificar la interfaz)

Si no, no es necesaria



# Principios SOLID (Robert Martin)

Única responsabilidad

Single Responsibility (SRP)

Abierto - cerrado

Open-closed (OCP)

Sustitución

Liskov Substitution (LSP)

Inversión de dependencia

Dependency Inversion (DIP)

Segregación de la interfaz

Interface Segregation (ISP)



# Única responsabilidad

Tiene que ver con la alta cohesión

Separar por

Uso

Razón de cambio



Por ejemplo: persistencia, ¿debe ser parte del comportamiento del objeto?

# Abierto - Cerrado

“Las clases tienen que estar cerradas para modificación, pero abiertas para reutilización”

=> No modificar las clases existentes

=> Extenderlas o adaptarlas por herencia o delegación



# Sustitución (Liskov)

En los clientes, debemos poder sustituir un tipo por sus subtipos

Relación “es un”

La subclase sea un subconjunto de la clase

Las clases base no deben tener comportamientos que dependan de las clases derivadas

O incluso de su existencia

Buscar el problema del círculo y la elipse en la Web



# Inversión de dependencia

No depender de clases concretas volátiles

No conviene que una clase herede o tenga una asociación hacia una clase concreta que tiene alta probabilidad de cambio

Las clases abstractas y las interfaces son más estables

Conviene utilizarlas para herencia o delegaciones

“Programe contra interfaces, no contra implementaciones”

En este caso estamos poniendo el foco en el uso de un cliente

```
Iterable c = new ArrayList( );  
Iterator i = c.iterator( );
```



# Segregación de la interfaz (1)

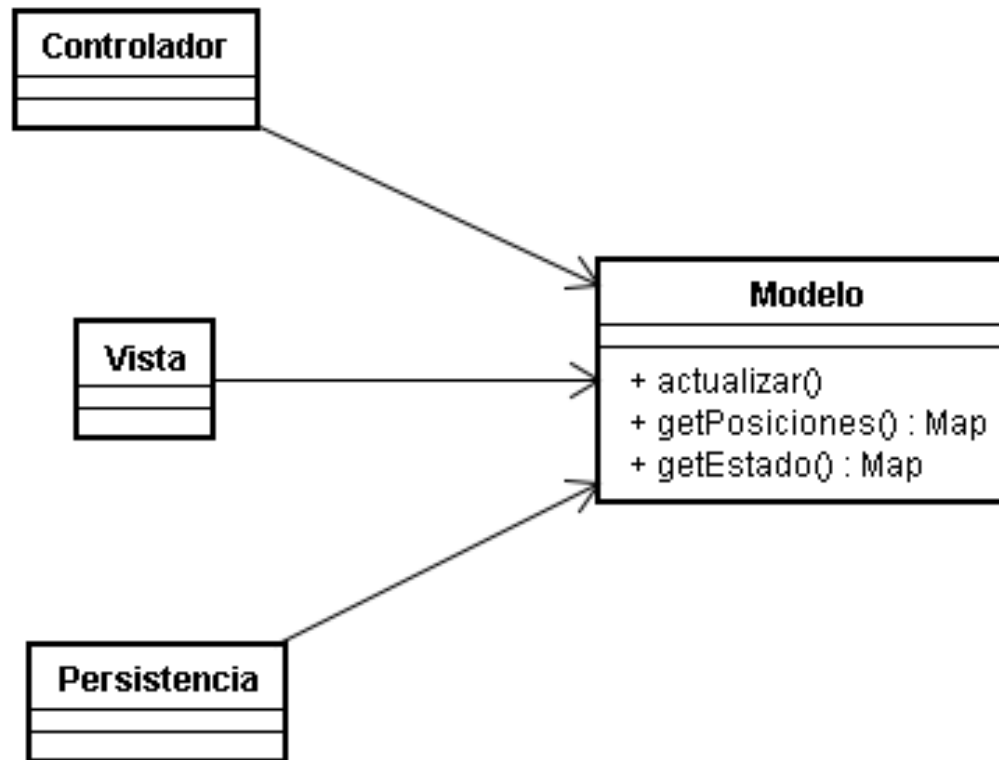
Los clientes de una clase no dependan de métodos que no utilizan

Si una clase tiene una referencia a, o hereda de, otra clase, de la cual sólo tiene sentido que utilice algunos de sus métodos, pero no todos, lo mejor sería separar la clase en cuestión en más de una

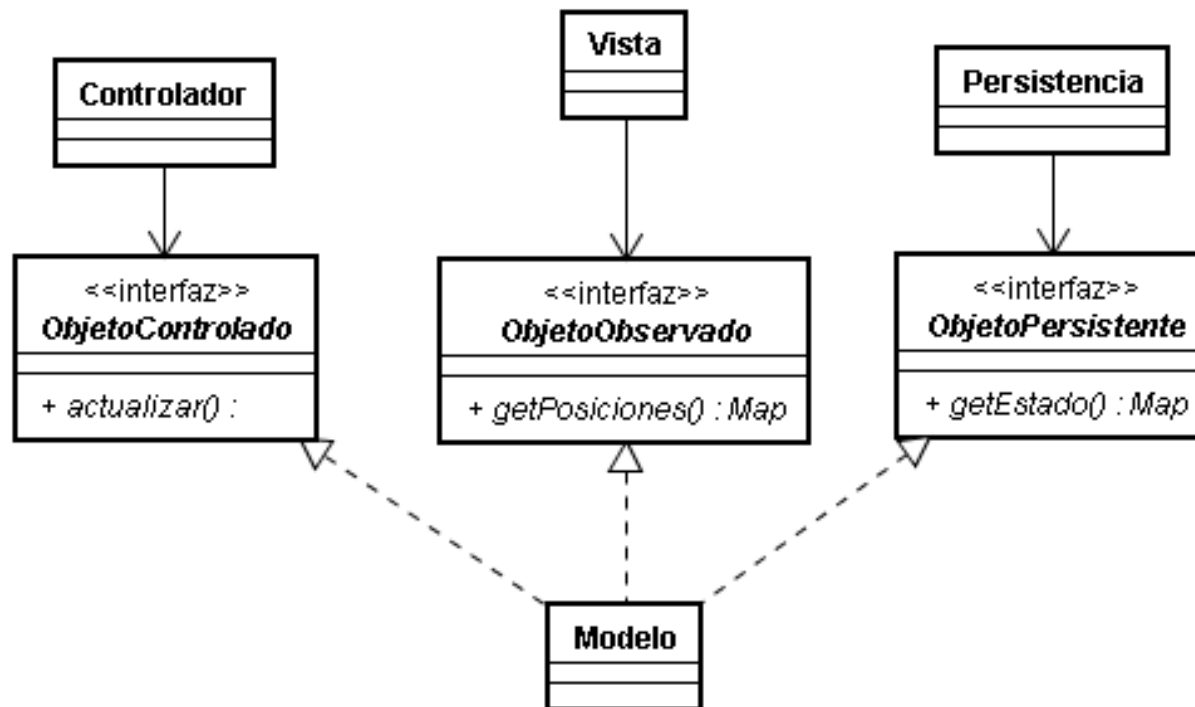
Se apoya en otros principios



## Segregación de la interfaz (2)



## Segregación de la interfaz (3)



Modelo implementa tres interfaces, y cada tipo de objeto la utiliza solamente a través de cierta interfaz

# Delegación sobre herencia

Herencia es estática

Reutilización de caja blanca: ¿viola el encapsulamiento?

Delegación otorga mayor flexibilidad

Permite diferir hasta el tiempo de ejecución el tipo de reutilización que se hará

Herencia (concreta, abstracta o de interfaces)

Cuando se va a reutilizar la interfaz

Delegación

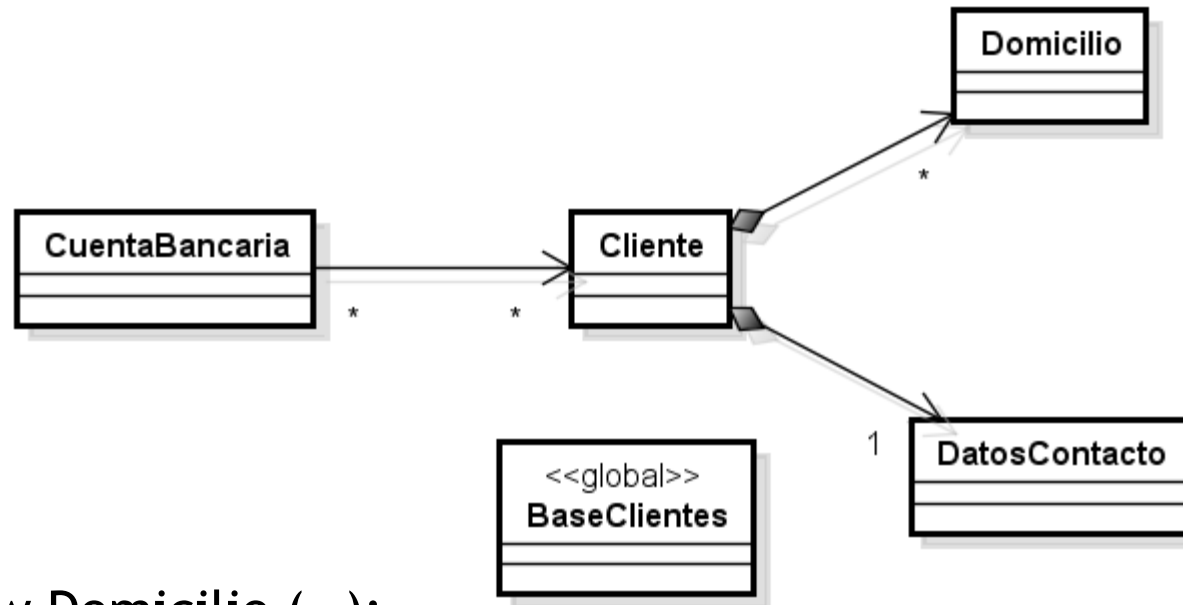
Cuando se van a reutilizar algunas responsabilidades (comportamiento o estado)

# Polimorfismo

## Evitar condicionales en POO

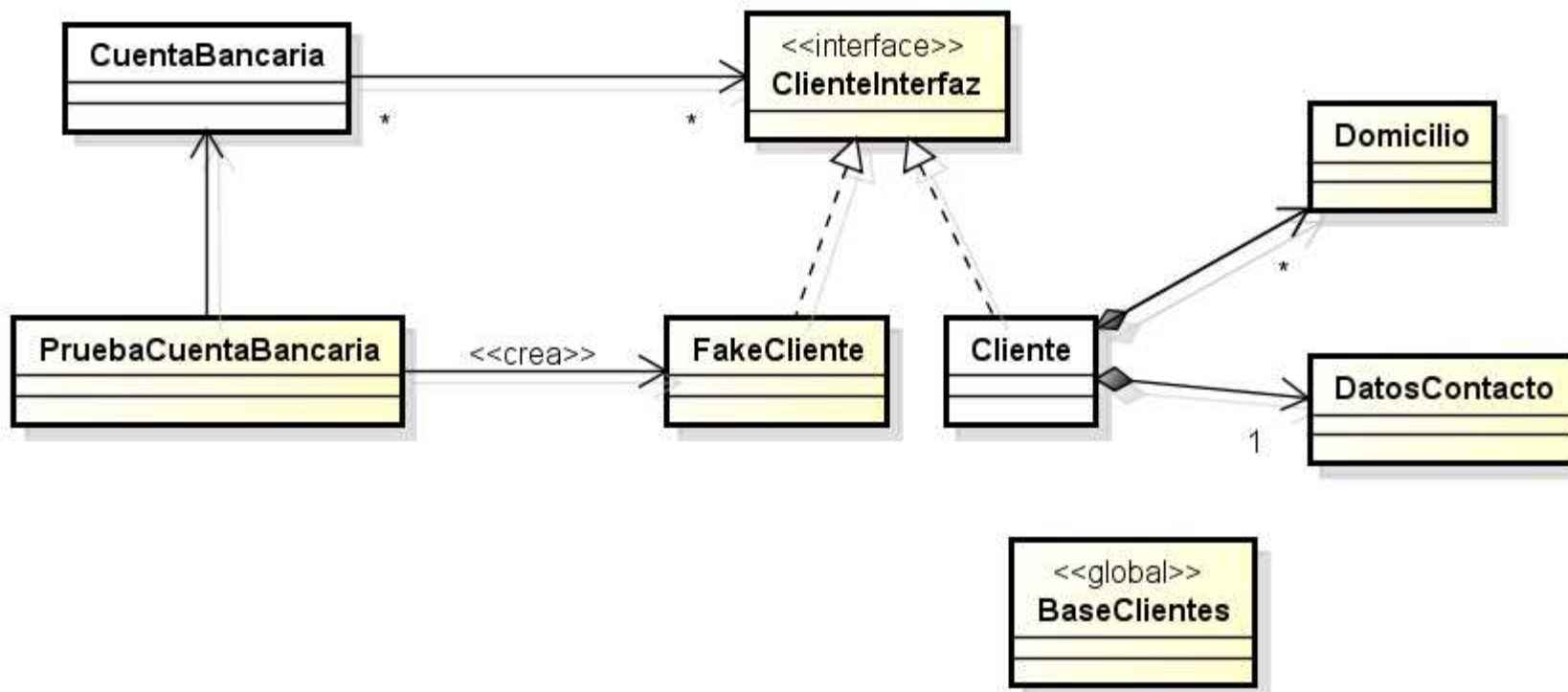
```
public void extraer (int monto) {  
    if (numero < 10000)                // caja de ahorro  
        if (monto > saldo) throw new SinSaldoException();  
        else saldo -= monto;  
    else                                // cuenta corriente  
        if (monto > saldo+descubierto)  
            throw new SinSaldoException();  
        else saldo -= monto;  
}
```

# Un caso: ¿cómo creo una instancia para probar?



```
domicilio = new Domicilio (...);
datosContacto = new DatosContacto(...);
if (!BaseClientes.contiene(dniCliente))
    cliente cliente = new Cliente (dni, ..., domicilio,
    datosContacto);
cuentaBancaria = new CuentaBancaria (cliente, ...);
```

# Mejor desacoplar



Así mantengo la prueba unitaria

Ya habrá tiempo para pruebas de integración...

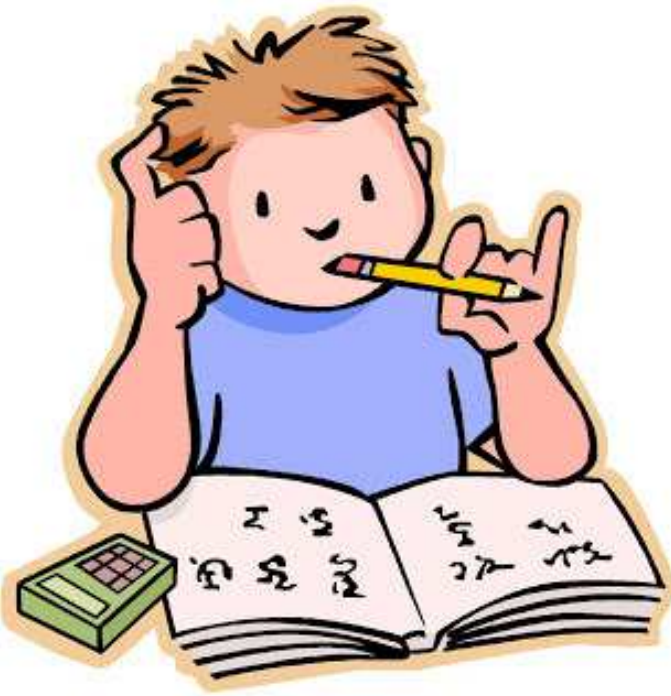
# Recapitulación



# Recapitulación: preguntas

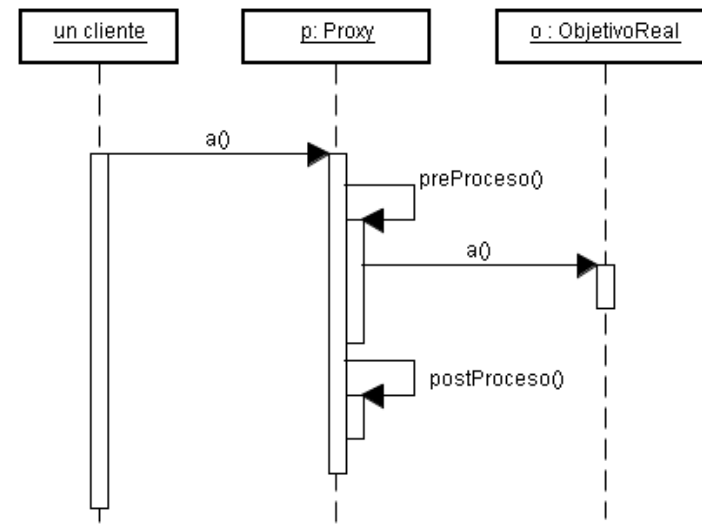
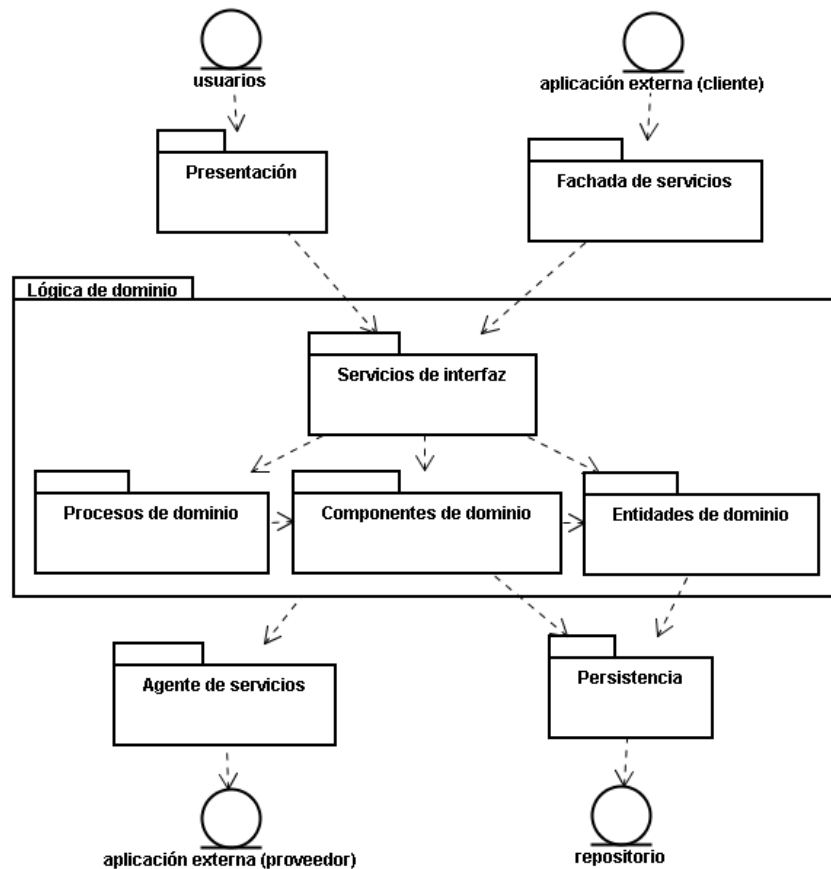
¿Por qué es preferible la delegación sobre la herencia?

¿Qué significa programar contra interfaces?



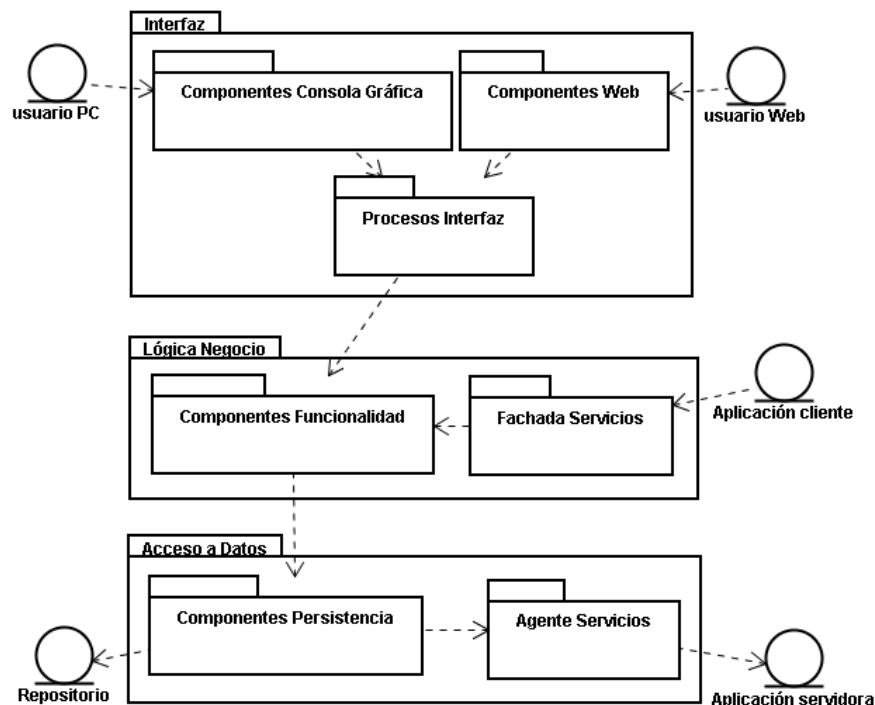


# Paquetes y métodos



# Diseño de paquetes (1)

Semánticamente menos relevantes que las clases  
Permiten un nivel de abstracción más alto  
Deben describir una visión más global de un sistema



## Diseño de paquetes (2)

### Bajo acoplamiento

Diagrama de paquetes de UML es una herramienta invaluable para analizar dependencias

Construir las agrupaciones de clases de modo tal que sirvan para reutilizarse

Las clases que se prevé que se van a usar conjuntamente deberían colocarse en el mismo paquete

Separar los paquetes por los clientes que los usan

Las clases que cambian poco debieran ubicarse en paquetes separados de aquellas que cambian mucho

# Cuándo crear métodos

## Situaciones típicas de refactorización

- Reducir complejidad

- Hacer legible un pedazo de código

- Simplificar tests booleanos

- Evitar duplicación de código

- Encapsular complejidad y dependencias de plataforma

- Mejorar cohesión

¡Siempre que genero métodos innecesarios para el cliente los hago privados!

Cada método debe tener un propósito simple

Mal ejemplo Java: `iterador.next();`

# Visibilidad

Todo debe ser lo más privado que se pueda

Principio de mínimo privilegio aplicado a clientes

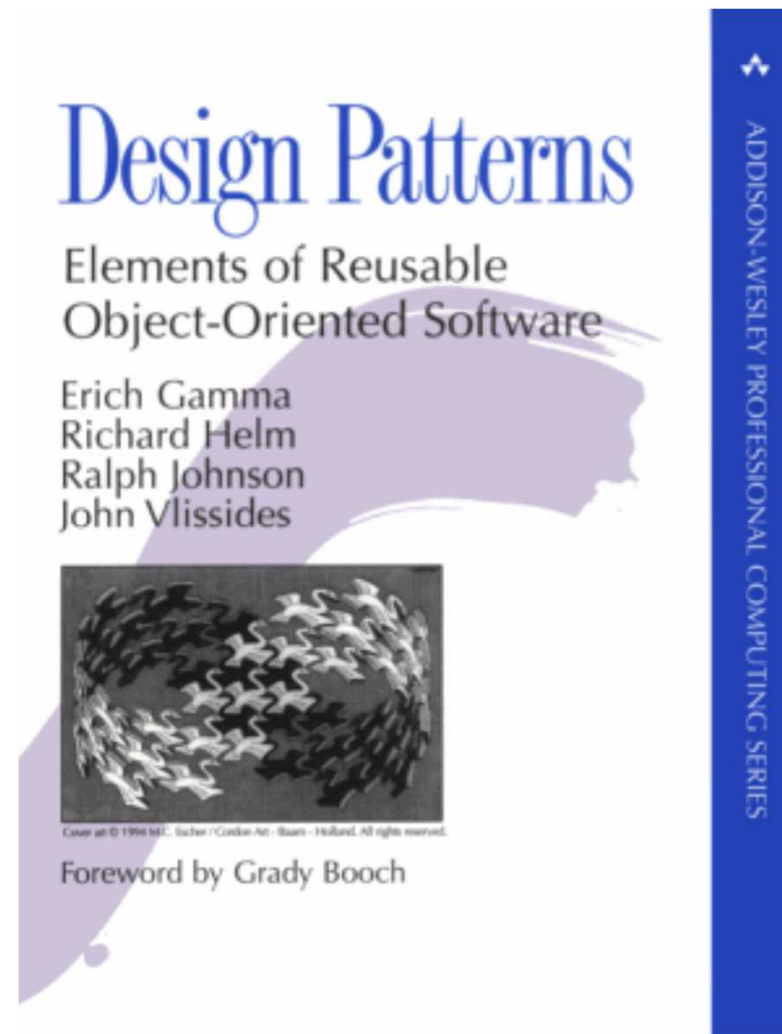
Garantiza que se pueda modificar código afectando al mínimo posible de clientes

Atributos privados

Métodos según lo que se requiera (no siempre públicos)

Propiedades, en general, públicas (pero el “get” o el “set” puede que no)

# Bonus: algo de patrones



fiuba  
algo3

# Primer patrón (1)

Ajedrez: cada tipo de pieza tiene un comportamiento distinto

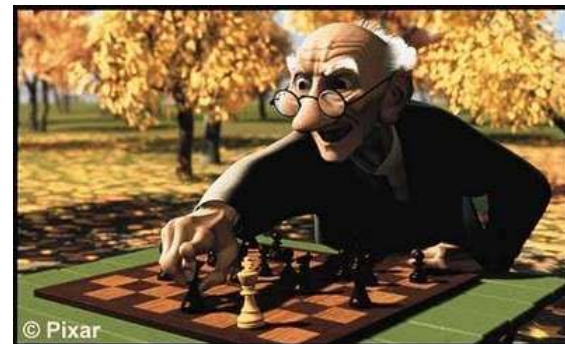
=> Implementación obvia (?)

Coronación: ¿qué ocurre?

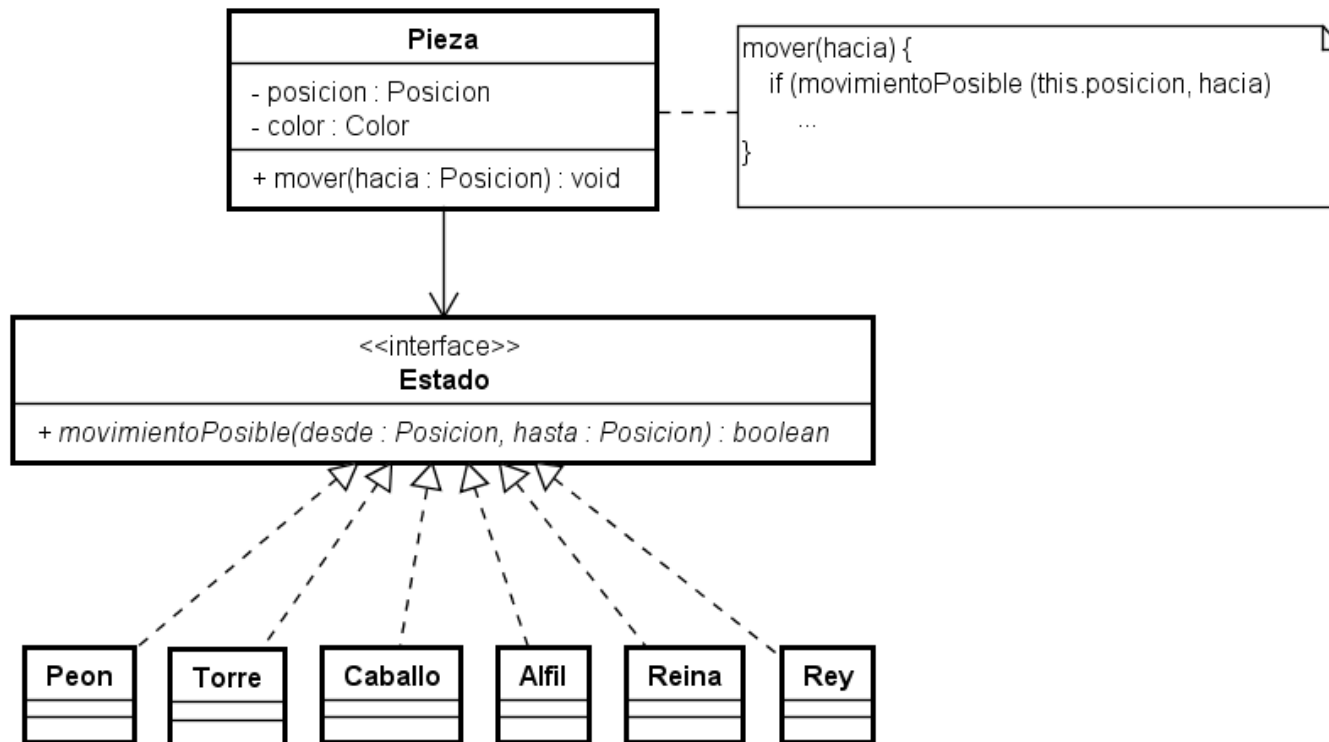
Posible interpretación: un objeto cambia de clase (su comportamiento varía luego de un cambio de estado)

... hay otros planteos: no es el mismo objeto

Quedémonos con el inicial: ¿puede un objeto cambiar su clase?



# Primer patrón (2)



Coronación: estado = new Reina ( );  
 ¿Cuántos objetos Torre, Alfil, etc., necesito?



# Intermedio: otro patrón (1)

¿Cuántos objetos Torre, Alfil, etc., necesito?  
No hay estado...

## Patrón Singleton

Una instancia para la clase  
Acceso global

Ojo que tiene desventajas  
=> Usar sólo si se ven ventajas



## Intermedio: otro patrón (2)

```
public class Alfil implements Estado {  
    private Alfil() { }  
    private static Alfil unicaInstancia = null;  
    public static Alfil getEstado () {  
        if (unicaInstancia == null)  
            unicaInstancia = new Alfil();  
        return unicaInstancia;  
    }  
    public boolean movimientoLegal (Posicion hacia) {  
        // TODO  
    }  
}
```

## Primer patrón (3)

Se llama “State”

Hemos encapsulado un método  
(*movimientoLegal*) en un objeto

Ocurre en otros patrones: Strategy, Command

Podemos tratar a los métodos como objetos

En Smalltalk todo es un objeto

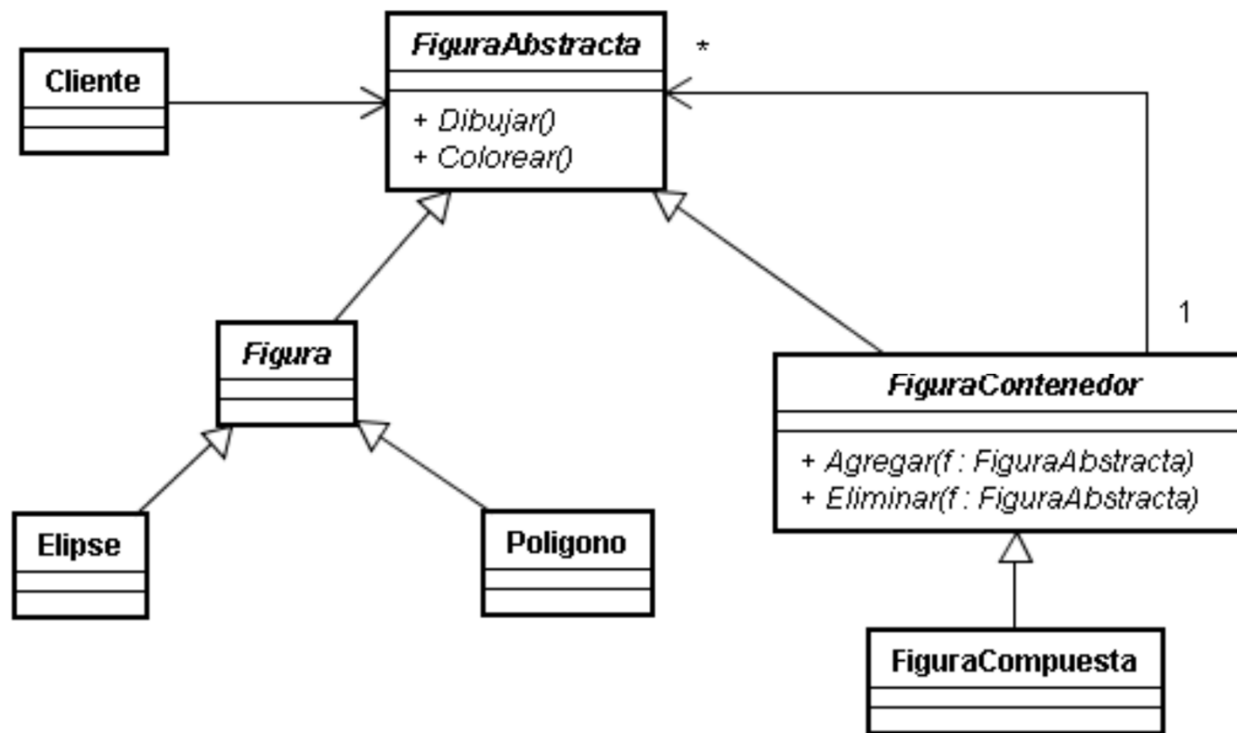
En C# hay “delegados”

Otra forma de verlo: cambia el algoritmo en  
tiempo de ejecución

Ahora el patrón se llama Strategy

# Composite: un patrón ¿sencillo?

Quiero tratar a un objeto y a una colección de objetos con la misma interfaz



## Otros patrones

Proxy: intermediario entre dos objetos

Adaptador: cambia la interfaz de una clase

Fachada: simplifica interfaz de varias clases

Observador: desacopla objetos observador  
y observado

Típico en MVC

Factory Method: encapsula la creación de  
un objeto

```
Iterator i = lista.iterator();
```

# Claves: los mandamientos del alumno de Algoritmos III (parte 2)

Mantendrás bajo el acoplamiento

Privilegiarás la delegación sobre la herencia

Preferirás las interfaces a las clases concretas

Darás la menor visibilidad posible a los clientes de tus clases

Mantendrás la entropía bajo control, modificando el diseño



# Lecturas obligatorias

Diapositivas de usabilidad / UX

En el campus

Code Complete, Steve McConnell, capítulo 5  
“Design in Construction”

Está en <http://cc2e.com/File.ashx?cid=336>



# Lecturas opcionales

UML para programadores Java, Robert Martin

Capítulo 6 “Principios de diseño OO”

Implementation Patterns, Kent Beck

Capítulos 5 a 9: casi todo el libro

Refactoring: Improving the Design of Existing Code, Martin Fowler

Capítulos 1 a 4

No están en la Web ni en la biblioteca

Orientación a objetos, diseño y programación, Carlos Fontela 2008, capítulos 16 y 17 “Diseño orientado a objetos: generalidades” y “Diseño orientado a objetos: principios y problemas típicos”



# Qué sigue

Temas avanzados derivados de POO

RTTI, reflexión

Lenguajes y POO

Tópicos de desarrollo de software

