Patrones de diseño

Sesión 1: Introducción y patrones básicos

Patrones de diseño

- En el desarrollo de aplicaciones J2EE (¡y no J2EE!) se presentan una y otra vez los mismos problemas
  - ¿Cómo estructuro el código de acceso a la b.d. para no cambiar demasiado si cambiamos de MySQL a Oracle?
  - ¿Cómo organizo los objetos de negocio para conseguir eficiencia en las llamadas a métodos en aplicaciones distribuidas?
- En lugar de solucionarlos siempre partiendo de cero, pueden utilizarse patrones de diseño
- Un patrón (*pattern*) es una solución **ya probada** y aplicable a un problema que se presenta **una y otra vez**

Un ejemplo de J2SE

- La mayoría conoceréis la forma de Java 1.4 de recorrer listas, los famosos *iterators*.

```
List lista;
...
Iterator it = lista.iterator();
while (it.hasNext()) {
   Object o = it.next();
   ...
}
```

- **Ventaja** fundamental: el bucle siempre es igual sea un ArrayList, un LinkedList, un Set, ...
- Es una buena idea, ¿no?..pero...¿es original de Java?

¡Pues no!

- No es ni más ni menos que el patrón **Iterator** (casualmente se llama igual y todo...).
- **Iterator** (según el GoF): proporciona una forma consistente de recorrer los *items* de una colección que es independiente de la colección subyacente.
  - El GoF define también los métodos que debería tener, entre los que se encuentra ***next*** e ***isDone*** (que es el *hasNext* de Java)

¿Por qué usar patrones?

- lo hacen hasta los diseñadores de lenguajes nuevos como Java: *no es necesario inventar la rueda si ya está inventada...*
- Son ideas **reutilizables**: se pueden usar una y otra vez
- Son expresivos: ayudan a crear un vocabulario común para el equipo de desarrollo
  - *(Desarrollador1)* ¡¡He tenido una idea fantástica, voy a usar un objeto que nos permitirá recorrer una colección independientemente de su implementación subyacente!!...Lo voy a llamar **desplazador**
  - *(Desarrollador2)* Ahh..te refieres a un iterador

Bibliografía

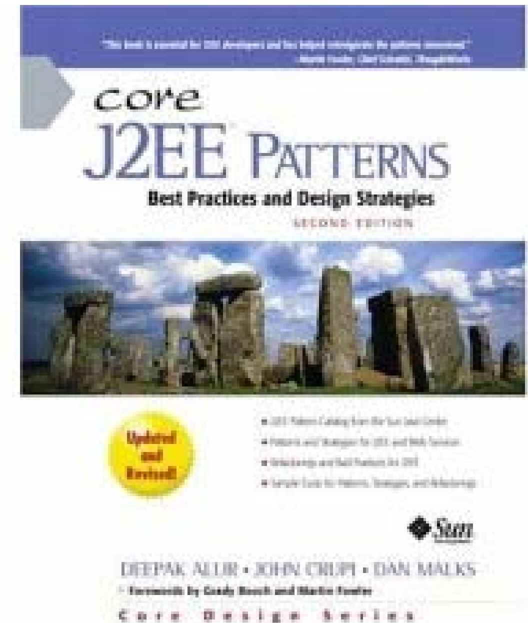
- Fueron popularizados por el libro “Design Patterns” por Gamma, Helm, Johnson y Vlissides (a partir de aquí, el “Gang of Four” [GoF])

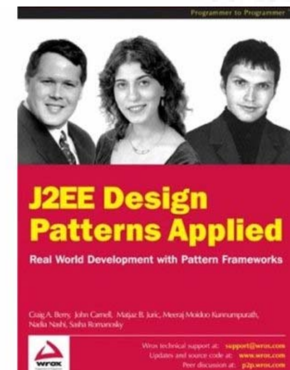
- Libro recomendado: “Head First Design Patterns”, Freeman & Freeman ed. O’reilly

© 2006-2007/ Depto. Ciencia de la Computación e IA

Patrones genéricos vs. JavaEE

- **Genéricos**: para cualquier aplicación y cualquier lenguaje de programación OO
- **J2EE/JavaEE**: especialmente pensados para las necesidades de aplicaciones web y enterprise en Java
  - Todo empezó con "*Core J2EE Patterns*"
  - Ahora hay más bibliografía específica





© 2006-2007/ Depto. Ciencia de la Computación e IA

Lo bueno y lo malo de los patrones

- Como todas las herramientas y metodologías los patrones no sirven para todo
  - Para un "hola mundo" probablemente no sean necesarios
- En general su idea es **añadir flexibilidad**: facilitar el cambio allí donde sea de esperar
- Esto suele ser a costa de **añadir complejidad** (capas al sistema, clases adicionales)
  - Para recorrer solo ArrayList no hace falta un iterador, basta con un for de los "de toda la vida"
- Solo el sentido común y la experiencia pueden indicar si un patrón es apropiado o no

¿Qué patrones vamos a ver?

- **¡No podemos verlos todos!:** hay 23 originales de “Design Patterns” más 21 de JavaEE más los que se inventa la gente en sus ratos de ocio…
- Muchos libros los clasifican por categorías (de creación, de responsabilidad, …) nosotros seguiremos un orden arbitrario, en función de nuestras necesidades de temario
- Para cada patrón diremos si es
  - GoF: aplicable a cualquier tipo de aplicación
  - JavaEE: aplicable a aplicaciones web y enterprise

© 2006-2007/ Depto. Ciencia de la Computación e IA

Hoy veremos…

- **Singleton** (GoF) : cómo crear un objeto único
- **DAO** (J2EE) : cómo aislar el acceso a la base de datos del resto de tareas
- **Transfer Object**: (J2EE) cómo empaquetar objetos para aumentar la eficiencia
- **Factory** (GoF): fabricación de objetos flexible
- **Facade** (GoF): interfaz simplificado para un sistema

© 2006-2007/ Depto. Ciencia de la Computación e IA

Singleton (GoF)

- Se utiliza cuando queremos asegurarnos de que en el sistema existe una única instancia de un objeto, y está inicializada de manera adecuada.

- Esto se podría hacer con variables static, pero…
  - ¿Qué ocurre si nos despistamos y repetimos la variable en sitios distintos?
  - Si unas dependen de otras, dependemos del orden de ejecución y/o compilación

© 2006-2007/ Depto. Ciencia de la Computación e IA

Diagrama de clases del singleton

- No tiene constructor público
- Las instancias se obtienen con getInstance() (en realidad siempre nos dará la misma)
- La instancia está almacenada dentro de la propia clase como una variable estática (retorcido, pero legal).

Singleton
- unicaInstancia : Singleton
+ getInstance() : Singleton

© 2006-2007/ Depto. Ciencia de la Computación e IA

“Idea feliz” para implementar un singleton

- Supongamos una clase con constructor privado

```
public class MiSingleton {
    private MiSingleton() {
        //aqui va el codigo del constructor
        ...
    }
}
```

- Solo podemos llamar al constructor desde un MiSingleton …pero ¿cómo obtenemos el PRIMER MiSingleton?

© 2006-2007/ Depto. Ciencia de la Computación e IA

"Idea feliz" para implementar un *singleton*

- Podríamos llamar al constructor a través de un método static

```
public class MiSingleton {
	private MiSingleton() {
		...
	}
		public static MiSingleton getInstance() {
			return new MiSingleton();
		}
	}
```

- Solo nos falta asegurarnos de que solo se crea una instancia

```
private static MiSingleton unico = null;
	public static MiSingleton getInstance() {
			if (unico == null)
		unico = new MiSingleton();
	return unico;
}
```

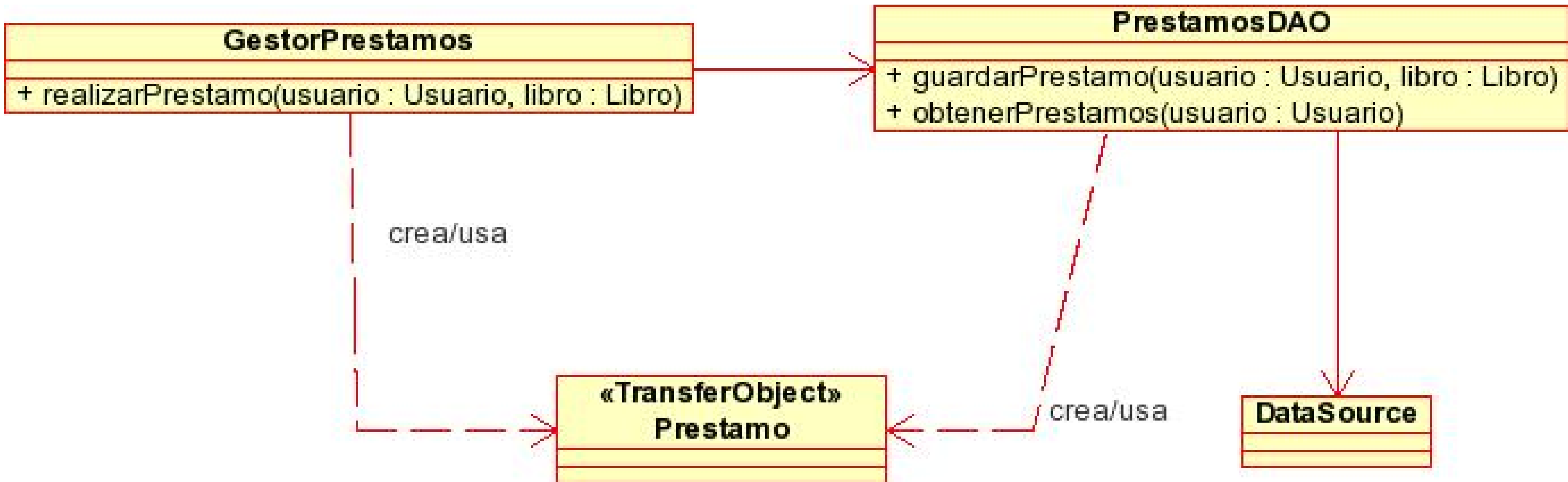

© 2006-2007/ Depto. Ciencia de la Computación e IA

Data Access Object (JavaEE)

- **Problema**: tenemos una clase que además de acceder a la base de datos tiene otras responsabilidades
- Ejemplo: GestorPrestamos de una biblioteca, implementa realizarPrestamo:
  - **Lógica de negocio**: número de días según tipo de usuario, no prestar si ya es moroso, etc.
  - **Acceso a datos:** crear un registro en la tabla “préstamos”
- **Solución**: crear una clase que se encargue **solo del acceso a datos**

© 2006-2007/ Depto. Ciencia de la Computación e IA

Diagrama de clases del DAO



© 2006-2007/ Depto. Ciencia de la Computación e IA

Beneficios básicos del DAO

- Separar el **acceso a datos** del resto de **funciones**

Separation of concerns: una clase no debe tener dos responsabilidades distintas

- **Independencia** del almacén de datos: sea base de datos relacional, fichero XML, .*properties*,… (con la ayuda del patrón *Factory*)

Hay que separar lo que permanece fijo de lo que puede variar en una aplicación

© 2006-2007/ Depto. Ciencia de la Computación e IA

Discusión de algunos aspectos

- El DAO no tiene por qué implementar todas las operaciones CRUD (Create-Read-Update-Delete)
- En general por cada objeto de negocio crearemos un DAO distinto (Libro->LibroDAO, Usuario->UsuarioDAO,…).
- El paso de información se encapsula en *Transfer Objects*

© 2006-2007/ Depto. Ciencia de la Computación e IA

Transfer Object (JavaEE)

- En bibliografía antigua aparece como Value Object (es lo mismo…)
- **Problemas**:
  - Evitar múltiples llamadas remotas en aplicaciones distribuidas
  - Pasar información de forma compacta

© 2006-2007/ Depto. Ciencia de la Computación e IA

Beneficios/problemas

- **Beneficio** fundamental: eficiencia
- **Problema**: viola un principio básico del diseño orientado a objetos

No se deben crear clases que no tengan comportamiento

© 2006-2007/ Depto. Ciencia de la Computación e IA

Discusión de algunos aspectos

- Los TOs suelen ser serializable
- Por cada objeto de negocio puede haber más de un TO:
  - **LibroBasicoTO**: almacena solo titulo y autor
  - **LibroDetalleTO**: almacena todos los datos de un libro
  - …
- Si los TOs son también de escritura hay que sincronizar la información

© 2006-2007/ Depto. Ciencia de la Computación e IA

Factory

- **Problema**: proporcionar una manera *flexible* de *instanciar objetos* cuando *la clase puede cambiar*, bien por cambios en el diseño, bien por cambios en tiempo de ejecución

```
Mensaje mensaje;
ICanal canal;
 ...
mensaje = GUI.getMensaje();
nombreCanal = GUI.getOpcionEnvio();
if (nombreCanal.equals("TCP"))
   canal = new EnvioTCP();
else if (nombreCanal.equals("SMS"))
   canal = new EnvioSMS();
else if (nombreCanal.equals("buzon"))
   canal = new EnvioBuzon();
canal.enviar(mensaje)
```


© 2006-2007/ Depto. Ciencia de la Computación e IA

Versión 1: *Simple Factory*

```
public class FactoriaCanales {
   public static ICanal crearCanal(String nombre) {
        ICanal canal;
        if (nombre.equals("TCP"))
              canal = new CanalTCP();
        else if (nombre.equals("SMS"))
              canal = new CanalSMS();
        else if (nombre.equals("buzon"))
              canal = new CanalBuzon();
        return canal;
   }
}
```

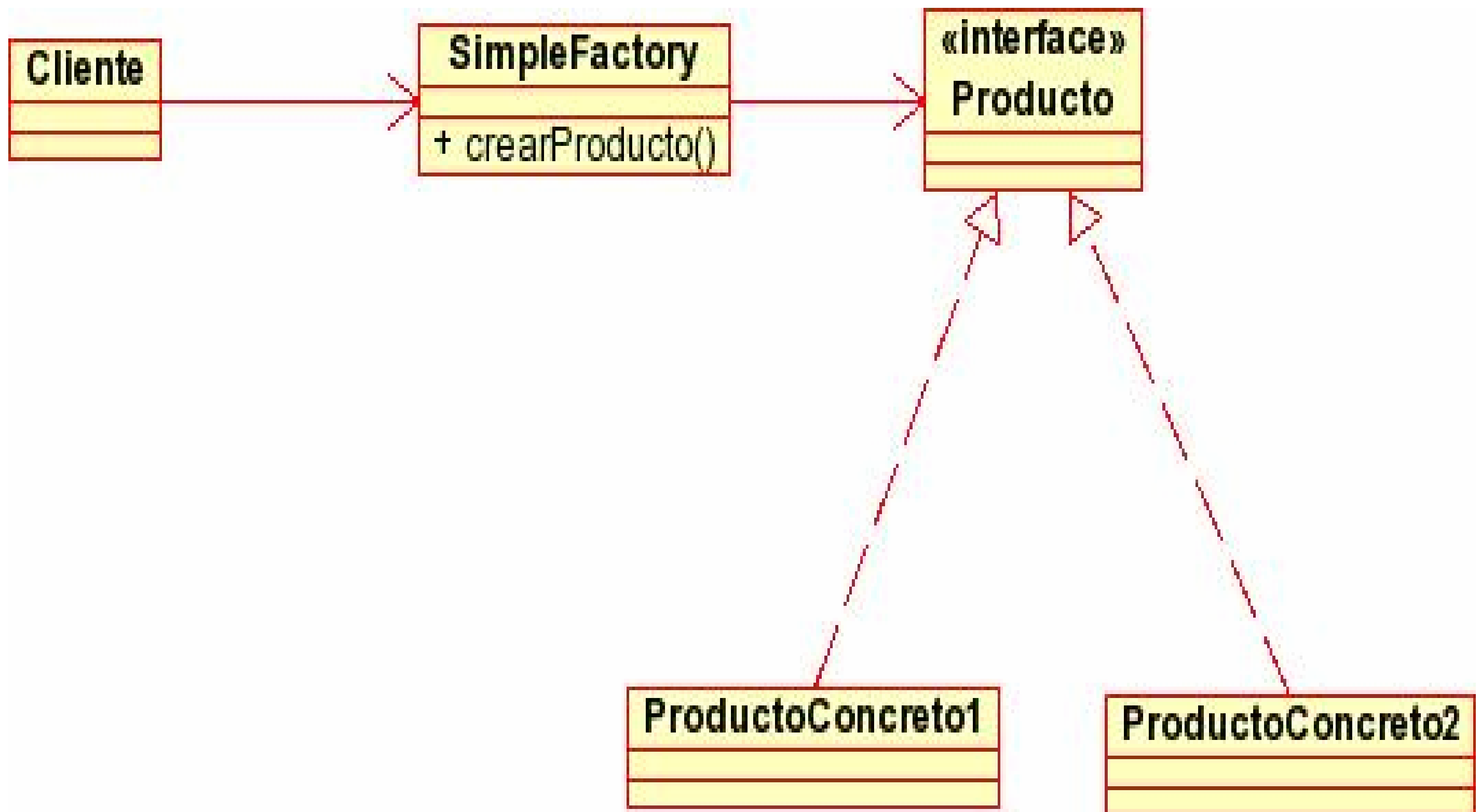
```
Mensaje mensaje;
ICanal canal;
...
mensaje = GUI.getMensaje();
nombreCanal = GUI.getOpcionEnvio();
canal = FactoriaCanales.crearCanal(nombreCanal);
canal.enviar(mensaje)
```

El principio abierto/cerrado: el código debe estar *abierto* a la extensión y *cerrado* a la modificación

Si este principio se redefiniera ahora probablemente diríamos "be water, my friend,…"

© 2006-2007/ Depto. Ciencia de la Computación e IA

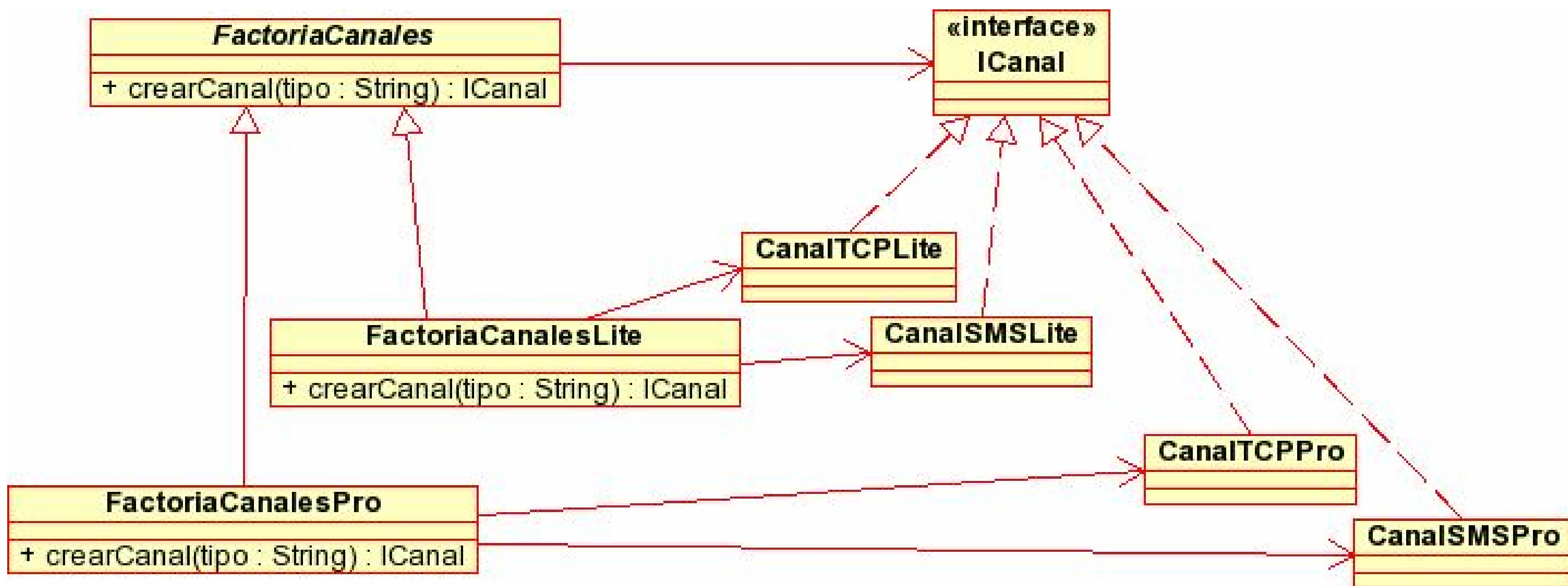
Diagrama de clases del *simple factory*



© 2006-2007/ Depto. Ciencia de la Computación e IA

Versión 2: *Factory method*

- Supongamos que se crean 2 tipos de usuarios, *lite* y *pro*, con distintas restricciones en los canales



© 2006-2007/ Depto. Ciencia de la Computación e IA

Beneficios del patrón:

- Hacéos las preguntas:
  - ¿Cómo se convierte un usuario *lite* en *pro*?
  - ¿Qué clases habría que crear para un nuevo tipo de usuario *silver*?
  - ¿Qué código habría que modificar?

© 2006-2007/ Depto. Ciencia de la Computación e IA

Diagrama de clases genérico del *Factory Method*

Factory

+ *crearProducto() : Producto*

«interface»
Producto

FactoryConcreta

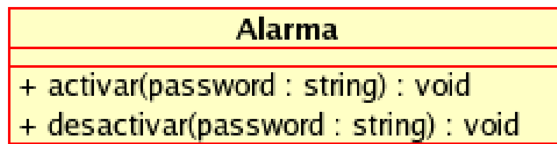
+ crearProducto() : Producto

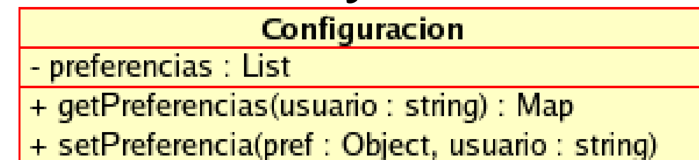
ProductoConcreto

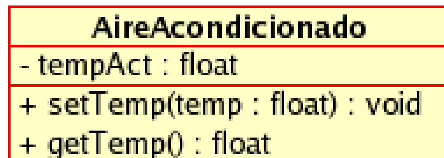
 © 2006-2007/ Depto. Ciencia de la Computación e IA

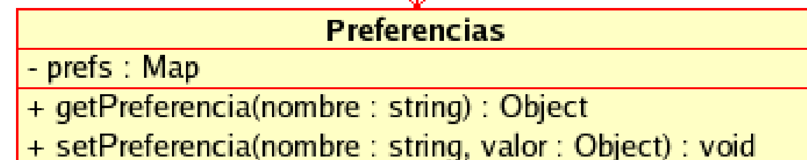
Facade (GoF)

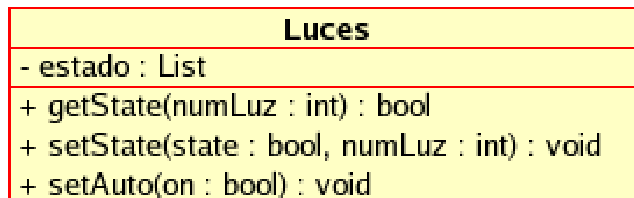
- Supongamos un sistema en el que realizar ciertas operaciones nos obliga a interactuar con un número elevado de clases y métodos











© 2006-2007/ Depto. Ciencia de la Computación e IA

Sin *facade*

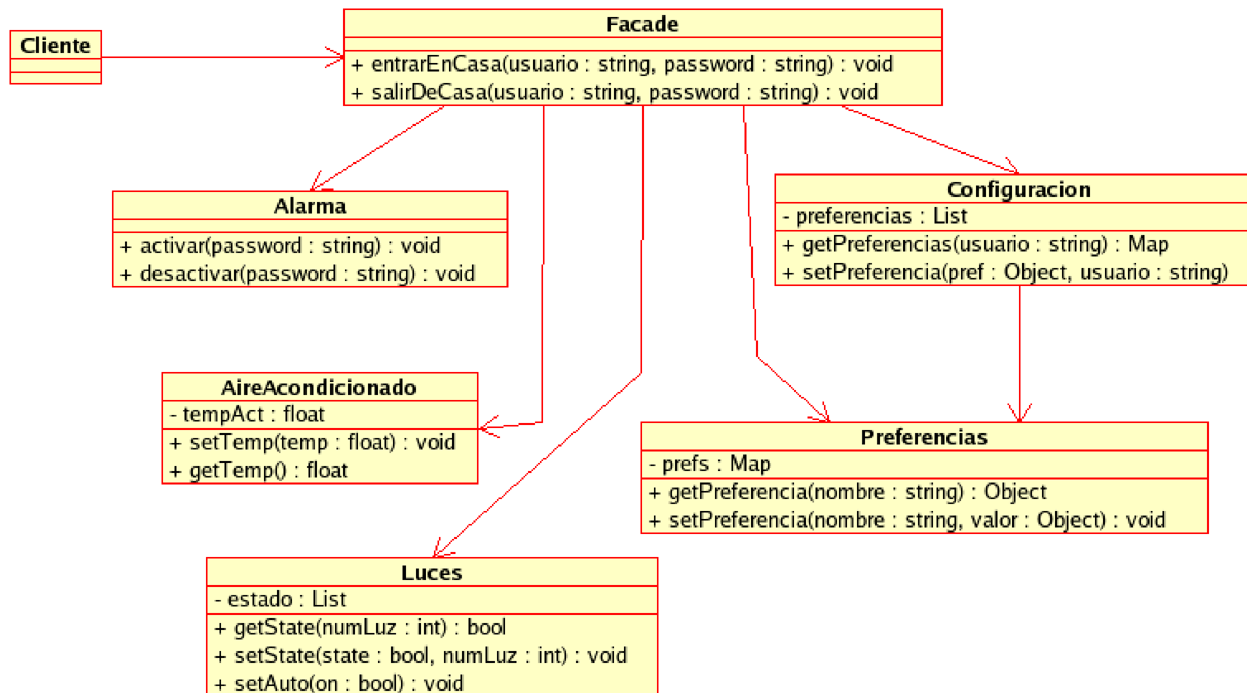
- entrarEnCasa()

```
//desactivar la alarma
alarma.desactivar(password);
//obtener preferencias de usuario
Preferencias prefs = configuracion.getPreferencias(usuario);
//poner el aire acondicionado a la temperatura deseada
Float temp = (Float) prefs.getPreferencia("temperatura");
aire.setTemp(temp.floatValue());
//poner las luces en "auto" si es la preferencia del usuario
String luces = (String) prefs.getPreferencia("luces");
if (luces.equals("auto"))
	luces.setAuto(true);
```


© 2006-2007/ Depto. Ciencia de la Computación e IA

Con *Facade*

- Clase que simplifica el interfaz del sistema, ofreciendo las operaciones más comunes



- El acceso “a bajo nivel” sigue siendo posible

© 2006-2007/ Depto. Ciencia de la Computación e IA

¿Preguntas...?