

INGENIERÍA DE SOFTWARE I

Unidad V

Tema: RUP Ágil.

Diseño de SW para el reuso II

Patrones GoF y GRASP 2^a parte

UNTDF – 2020

Agenda

- Uso de patrones en la Fase de Elaboración.
- Por qué usar Patrones de Diseño
 - Repaso de Conceptos
 - Ejemplo
- Patrones GoF:
 - Strategy
 - Patrón Adapter
 - Patrón Facade
- Patrones GRASP 2º Grupo
 - Polimorfismo
 - Fabricación Pura
 - Indirección
 - Variaciones Protegidas

Fase de Elaboración: La Segunda Iteración

- Durante la Fase de Inicio y la primera iteración de la Fase de Elaboración, el énfasis se puso en las actividades fundamentales de análisis y diseño.
- En la Segunda Iteración, el énfasis se pondrá en el refinamiento del Modelo de Diseño:
 - El uso de Patrones de diseño para crear una solución de diseño sólida



Aplicando Patrones durante el Diseño:

- Patrones GRASP
- Patrones Gof

Por qué usar Patrones de Diseño

- Encontrar los objetos de Diseño apropiados
- Determinar la granularidad de Objetos
- Especificar Interfaces de Objetos
- Especificación de la Implementación de Objetos
 - Programar a una Interface, no a una implementación
- Incrementar reusabilidad
 - Herencia vs Composición
 - Delegación
- Soporte de Extensibilidad
 - Frameworks

Patrones GoF: Gang of Four

- Hemos usado un Patrón GoF, Catálogo de diferentes patrones de diseño.
- Tres tipos diferentes
 - *Creacionales* – “creando objetos de una manera adecuada a la situación”
 - *De Estructura* – “facilitando el diseño mediante la identificación de formas simples de definir relaciones entre objetos”
 - *De comportamiento* – “patrones de comunicación comunes entre objetos”

Por qué Patrones? Ejemplo

■ Ejemplo: Juego de Simulación *SimUPato*

- El juego muestra una variedad de especies de Pato nadando y haciendo quack.
- Hay una pantalla con un estanque
- Hay botones para Quack, Nadar, etc.
- Se puede apretar un botón para “Todos hacen quack”, “Todos nadan”, etc.
- Un Pato puede:
 - nadar
 - quack
 - display
- Se asume que todos los patos nadan y hacen quack igual, difiere su display

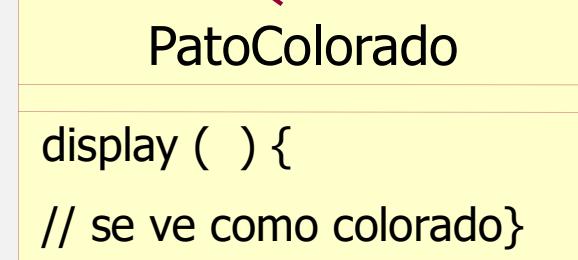
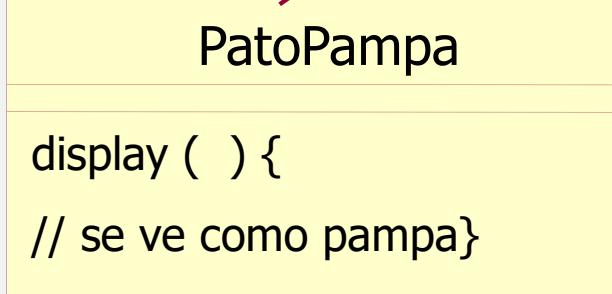
Por qué usar Patrones de Diseño. Ejemplo

Ejemplo. Jerarquía de Patos

■ Jerarquía de Patos



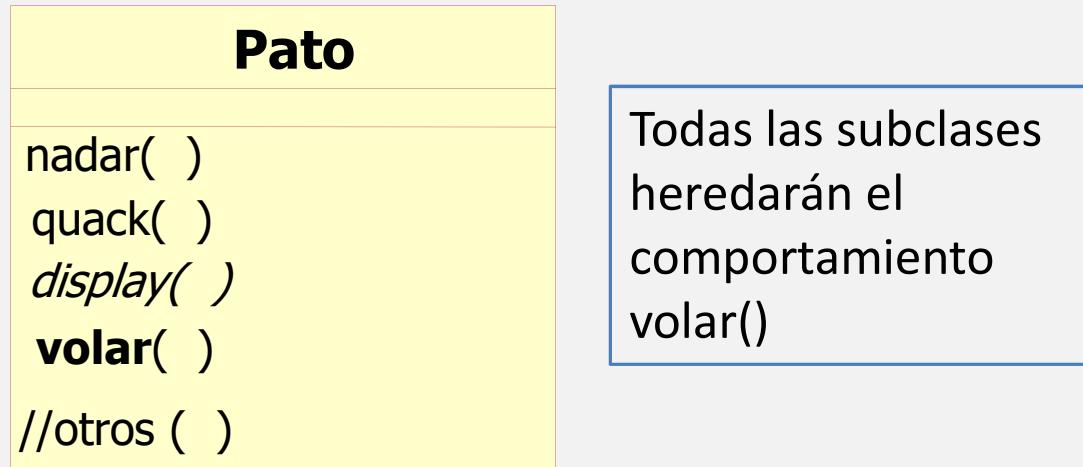
Todos los patos nadan y hacen quack
La superclase Pato tiene el código de implementación de quack y nadar. *display()* se implementa en las subclases



Cada subtipo de Pato es responsable de implementar su propia *display()*

Ejemplo. Se agregan requerimientos...

- **Nuevo requerimiento!!!:** Agregar el comportamiento *volar()*
 - Se agrega el método *volar()* a la superclase Pato



- *Problema: los patos de goma no vuelan!!!*
- Sin embargo la clase PatoGoma hereda el método *volar()* de Pato. Cuando se hace click en TodosVuelan, el PatoGoma sale volando!!!

Ejemplo. Se agregan requerimientos...

- Cómo puede resolverse este problema?
 - Se podría override volar() en **PatoGoma** para que no haga nada
 - Además PatoGoma no hace quack, sino que chilla.

PatoGoma

```
quack( ) { //chilla}  
display( ) { //se ve de goma}  
volar( ) // override para que  
no haga nada}
```

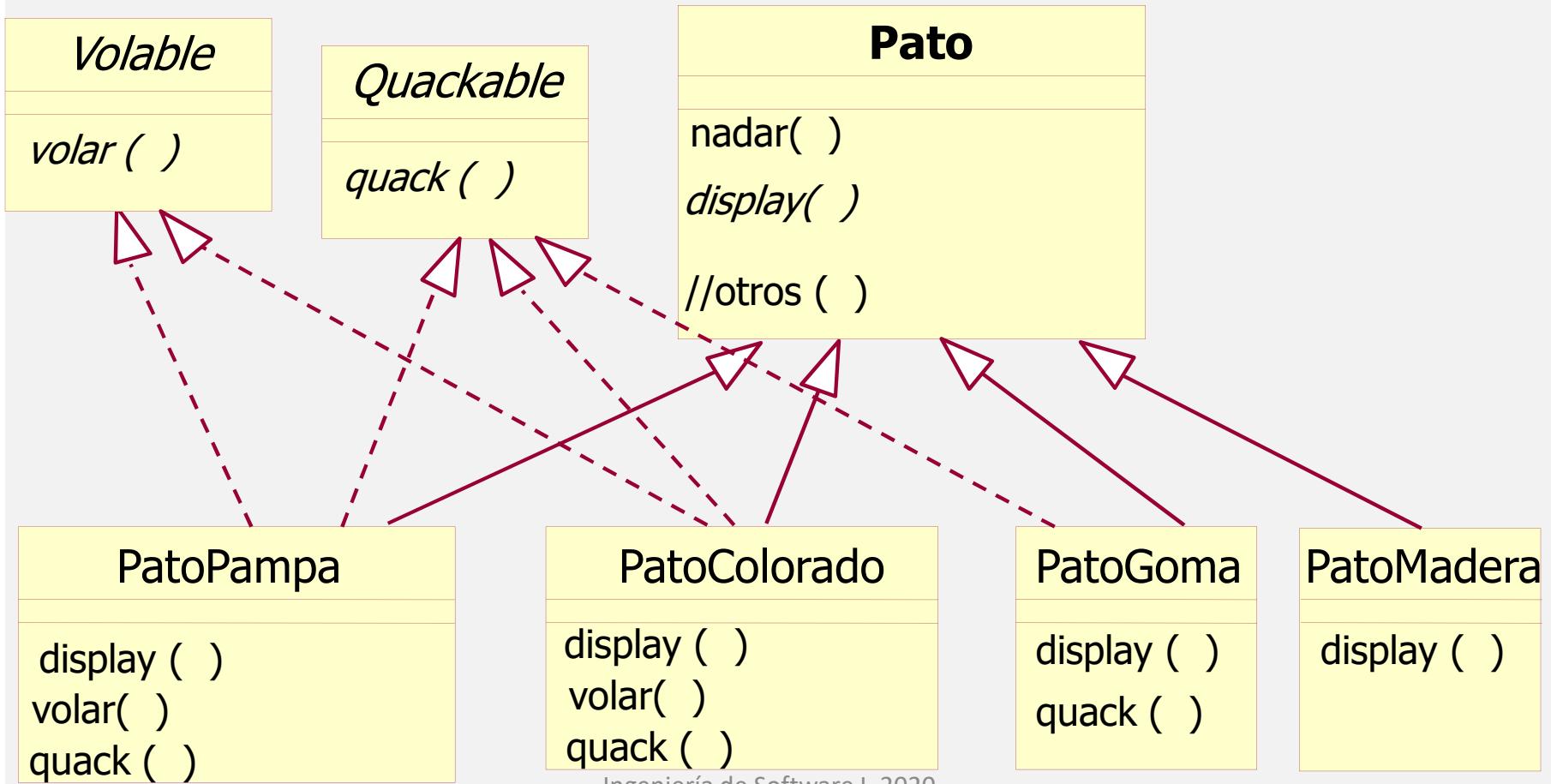
- Si agregamos a la Jerarquía **PatoMadera**, no hace quack, ni chilla, habría que sobreescribir quack para que no haga nada.
- **Se toma conocimiento que se pretende cambiar el juego c/6 meses**

Ejemplo. Requerimientos cambiantes

- **Problema de requerimientos cambiantes**
- Se podría usar Herencia para abstraer diferentes comportamientos, pero sería complicado cada vez que los requerimientos cambien
- *Usando herencia, vemos que no todas las clases derivadas usan el comportamiento de la clase base. La Herencia implica muchas decisiones en tiempo de compilación*
- Lo que parecía bueno para el reuso no lo fue para el mantenimiento
- *Se piensa en una solución alternativa, usando sólo interfaces para volar() y quack() que sea implementada por las clases que utilicen estos métodos*

Ejemplo. Solución con Interfaces

- Solución usando interfaces para `volar()` y `quack()` que son los comportamientos cambiantes



Ejemplo. Problemas de las soluciones

■ El problema.

- *La Herencia no funcionó*
 - No todas las subclases necesitan volar o hacer quack
 - Hacer un cambio en la superclase causa problemas en el mantenimiento
- *Las Interfaces Volable/Quackable no funcionaron*
 - Tengo que repetir código de volar y quack de todas las subcasetes de Pato. Se destruye el reuso de código que se puede compartir
 - Además pueden existir diferentes comportamientos Volar/Quack entre los patos que vuelan

Ejemplo. Principio de Diseño 1

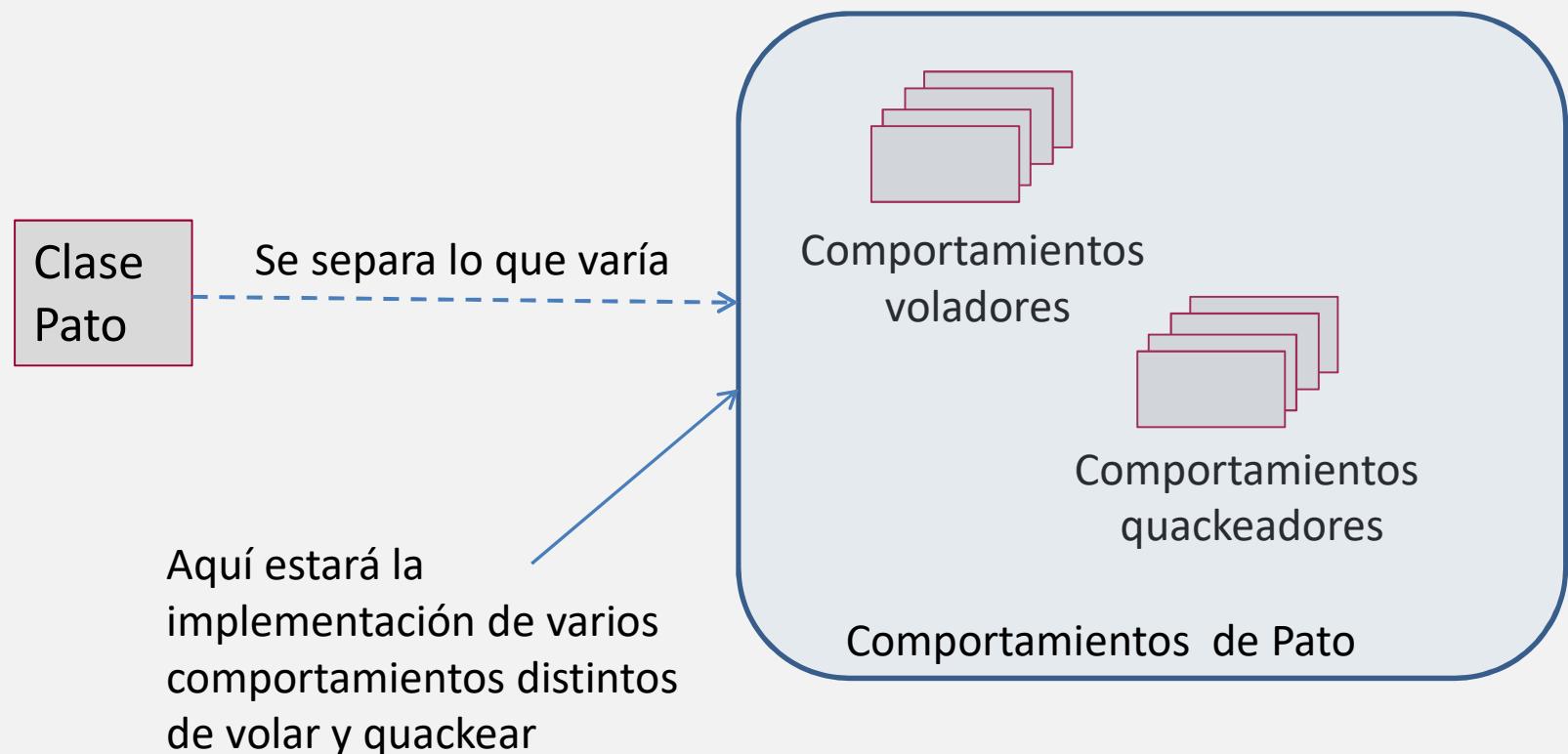
- **Principio de Diseño:** “*Identificar los aspectos de una aplicación que pueden variar y separarlos de los que se mantienen iguales*”
 - Se puede alterar o extender las partes que varían sin afectar las partes que son estáticas.
 - **Volviendo a SimUPato**
 - Qué partes varían?
 - Qué partes no varían?
 - Cómo podemos sacar el comportamiento que varía de la clase Pato?
 - Y cómo podemos extender este comportamiento en el futuro sin tener que reescribir el código existente?
- 
- Partes que varían: Volar y hacer Quack
 - Partes que se mantienen igual: Display y Nadar

Ejemplo. Separando lo que cambia

- Sabemos que volar() y quack() son las partes de la clase Pato que varían en distintos tipos de patos
- Separamos estos comportamientos de la clase base Pato.
 - Creamos dos conjuntos de clases (uno para Volar y otro para hacer Quack).
 - Estos dos conjuntos representan el comportamiento que varía.

Ejemplo. Separando lo que cambia

- Sacamos de la clase base Pato los comportamientos que pueden variar (quack y volar) y los ponemos en otra estructura de clases



Ejemplo. Comportamientos flexibles

- Cómo diseñamos el conjunto de clases que implementan los comportamientos Volar y Quack?
- Queremos que los comportamientos sean flexibles y sabemos que queremos asignar comportamiento a las instancias de Pato:
 - *Por ejemplo: Crear un PatoColorado que pueda volar y hacer quack. Pero luego, con un simple cambio de código, cambiar PatoColorado, de forma que no pueda volar y cuando haga quack cante una melodía.*
- Dadas estas metas, se establece un segundo Principio de Diseño

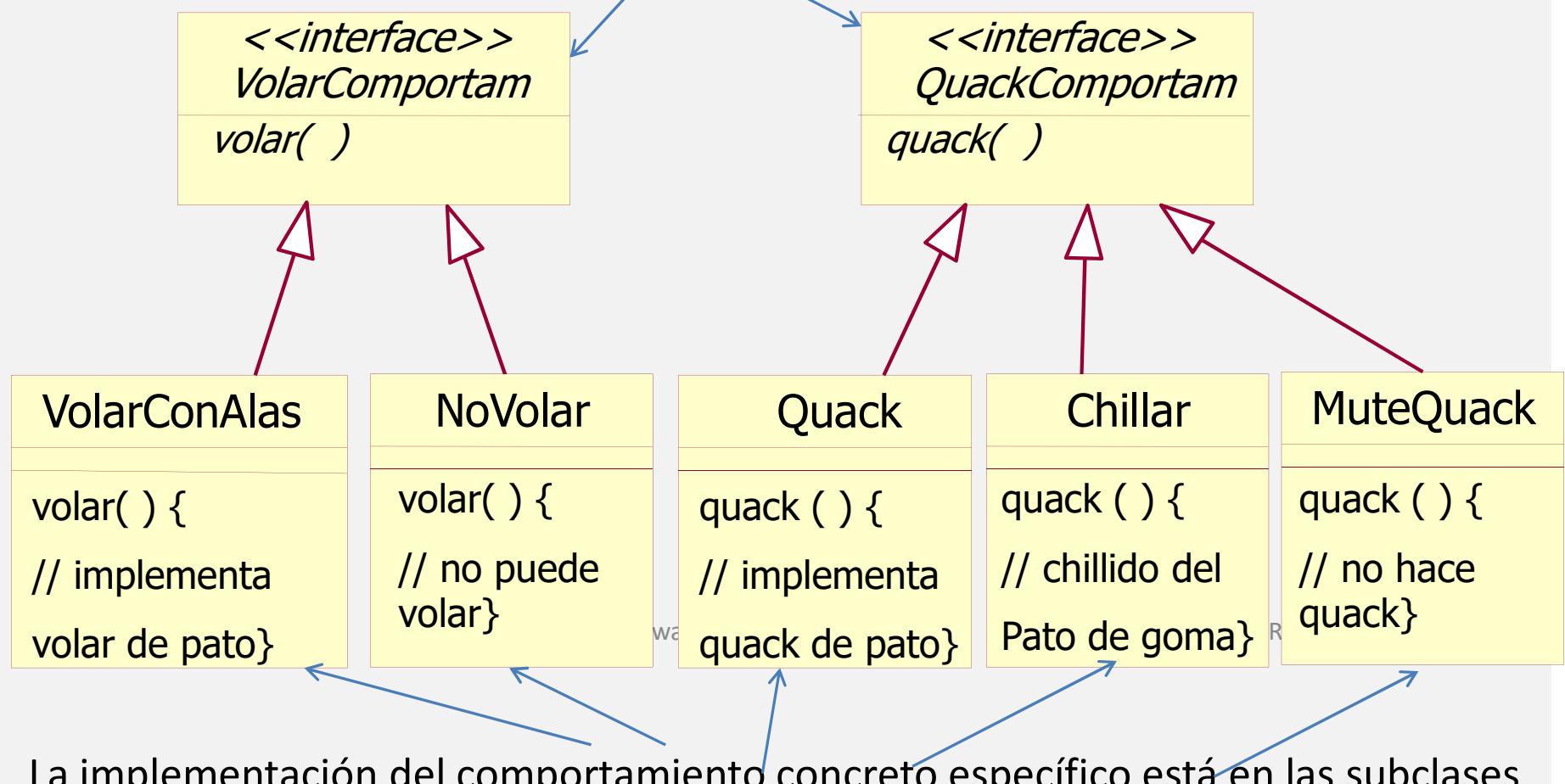
Ejemplo. Principio de Diseño 2

- ***Programar a una interface, no a una implementación.***
 - (Mejor “Programar a una abstracción, no a una implementación”)
- Si definimos las **interfaces VolarComportam** y **QuackComportam**, la clase Pato quedará vinculada a estas interfaces, pero no a una implementación específica.
- Cada una de estas interfaces (**VolarComportam** y **QuackComportam**) será implementada en su propia jerarquía (en lugar de ser implementada por las subclases de Pato)
- Cada subclase de Pato usará una implementación específica de ese comportamiento.

Por qué usar Patrones de Diseño. Ejemplo

Ejemplo. Nuevo Diseño de Interfaces

- Nuevo diseño: Las subclases de Pato usarán el comportamiento representado por las interfaces



Ejemplo. Beneficios del diseño elegido

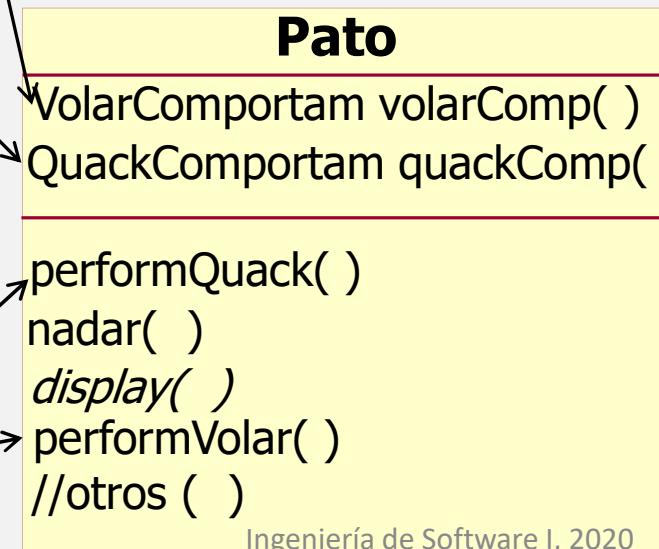
- **Beneficios del Nuevo diseño:**
- Con el nuevo diseño otros tipos de objetos pueden reusar los comportamientos volar y quack, porque ya no están escondidos en las subclases de Pato
- Podemos agregar nuevos comportamientos sin modificar ninguna de las clases existentes o tocar cualquiera de las clases Pato que usan dichos comportamientos.

Ejemplo. Integrando...

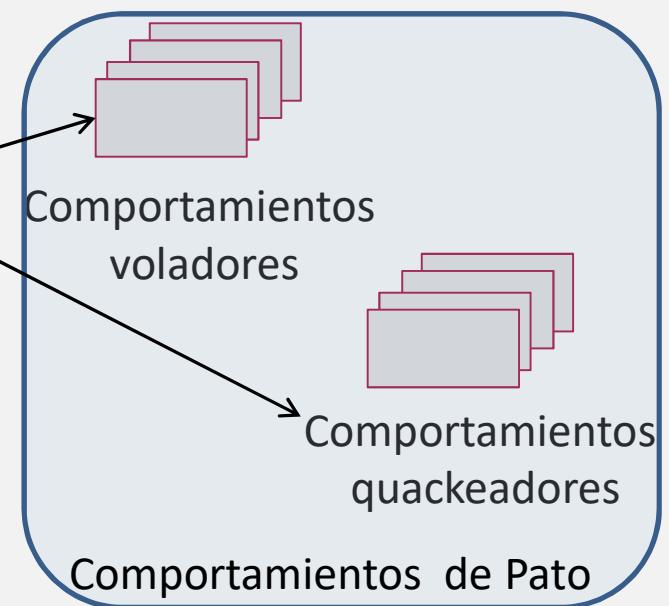
■ Integrando la jerarquía Pato al Diseño:

- La clave es que un Pato ahora delega su comportamiento “volar” y “quackear”, en lugar de usar los métodos definidos en Pato o sus subclases.
- Se agregan 2 variables de instancia a la clase Pato que mantienen una referencia a su comportamiento específico en run-time

Las variables se declaran del tipo de la Interface de comportamiento



Estos métodos reemplazan a volar y quack



Ejemplo. Delegando comportamiento

- Implementamos **performQuack()**:

```
public class Pato {  
    QuackComportam quackComp; ← Cada Pato tiene una referencia a algo que  
    //....  
    public void performQuack() {  
        quackComp.quack; ← El Objeto Pato delega este comportamiento al  
    }  
}
```

- Hasta aquí, hemos realizado la Delegación a la Jerarquía de comportamiento apropiada, pero todavía no hemos definido qué comportamiento específico que queremos.

Ejemplo. Definiendo el comportamiento

- **Cómo se setean las variables de instancia que definen el comportamiento específico:**

```
public class PatoColorado extends Pato {  
    public PatoColorado() {  
        quackComp = new Quack(); ←  
        volarComp = new VolarConAlas();  
    }  
}
```

PatoColorado hereda las variables de instancia de Pato

El Objeto PatoColorado usa la clase Quack para manejar su graznido y VolarConAlas para su vuelo. Cuando ejecute performQuack, se delegará al comportamiento apropiado

- Cuando PatoColorado se instancie, su constructor inicializa la variable de instancia de PatoColorado quackComp, (heredada de Pato), a un tipo Quack (Clase concreta de implementación de QuackComportam).
- Lo mismo para VolarComp.

Ejemplo. Definiendo el comportamiento

- Una vez implementadas las clases e interfaces, se hace el test

```
public class MiniPatoSimulator{  
    public static void main(String[] args) {  
        Pato colorado = new PatoColorado();  
        colorado.performQuack();  
        colorado.performVolar();  
    }  
}
```

- Resultado del test:

Quack
Estoy volando!

Ejemplo. Cambiando el comportamiento

- Definiendo el comportamiento dinámicamente:
 1. *Se agregan métodos a la clase Pato para modificar las variables de instancia*

```
public void setVolarComp(VolarComportam vC) {  
    volarComp = vC;  
}  
  
public void setQuackComp(QuackComportam qC) {  
    quackComp= qC;  
}
```

Se agregan estos dos métodos a la clase Pato

- Podemos llamar estos métodos en cualquier momento para cambiar el comportamiento.

Ejemplo. Agregando un tipo de Pato

- Definiendo el comportamiento dinámicamente:
 2. *Se crear un nuevo tipo de Pato (PatoModelo)*

```
public class PatoModelo extends Pato {  
    public PatoModelo() {  
        quackComp = new Quack();  
        volarComp = new NoVolar();  
    }  
    public void display(){  
        System.out.println("Soy un Pato modelo");  
    }  
}
```

Ejemplo. Nuevo comportamiento volar

- Definiendo el comportamiento dinámicamente:
 3. *Se crea un nuevo tipo de comportamiento Volar*

```
public class VolarEnCohete implements VolarComportam {  
    public void volar() {  
        System.out.println("Estoy volando con un cohete!");  
    }  
}
```

Ejemplo. Haciendo un test del nuevo Pato

- Definiendo el comportamiento dinámicamente:

4. Cambiamos la clase test

```
public class MiniPatoSimulator{  
    public static void main(String[] args) {  
        Pato colorado = new PatoColorado();  
        colorado.performQuack(); // creado con el comportamiento Quack  
        colorado.performVolar(); //creado con el comportamiento Volar  
  
        Pato modelo = new PatoModelo();  
        modelo.performVolar(); //lo había creado con el comportamiento NoVolar  
        modelo.setVolarComp(new VolarEnCohete()); //cambio el comp. dinámicamente  
        modelo.performVolar();  
    }  
}
```

Ejemplo. Haciendo un test del nuevo Pato

- Definiendo el comportamiento dinámicamente:
- Resultado del test

Quack

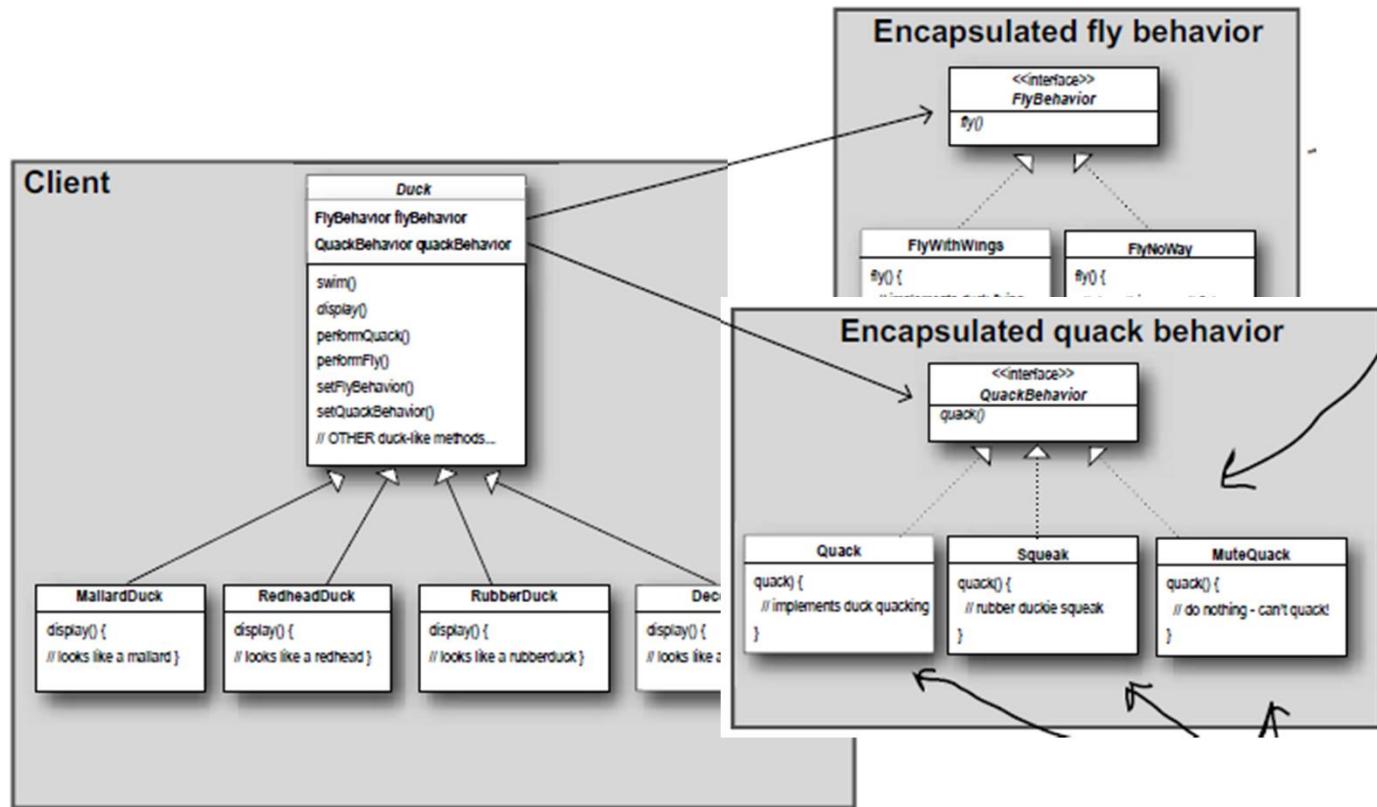
Estoy volando

No puedo volar

Estoy volando con un cohete!

Por qué usar Patrones de Diseño. Ejemplo

Ejemplo. El nuevo Diseño integrado



Ejemplo. Principio de Diseño 3

- **Principio de Diseño 3: Favorecer la composición sobre la herencia**
- La relación **tiene-un** es mejor que la relación **es-un**
- Cada Pato tiene-un VolarComportam y un QuackComportam a los cuales les delega la responsabilidad de volar y quackear
- En lugar de heredar comportamiento, los patos se componen con el objeto comportamiento apropiado.
- El uso de composición brinda gran flexibilidad al diseño.
 - Permite encapsular una familia de algoritmos en su propio conjunto de clases (puede ser reusado)
 - Permite cambiar comportamiento en run-time
- La composición se utiliza en muchos patrones de diseño

Ejemplo. Pensemos en algoritmos

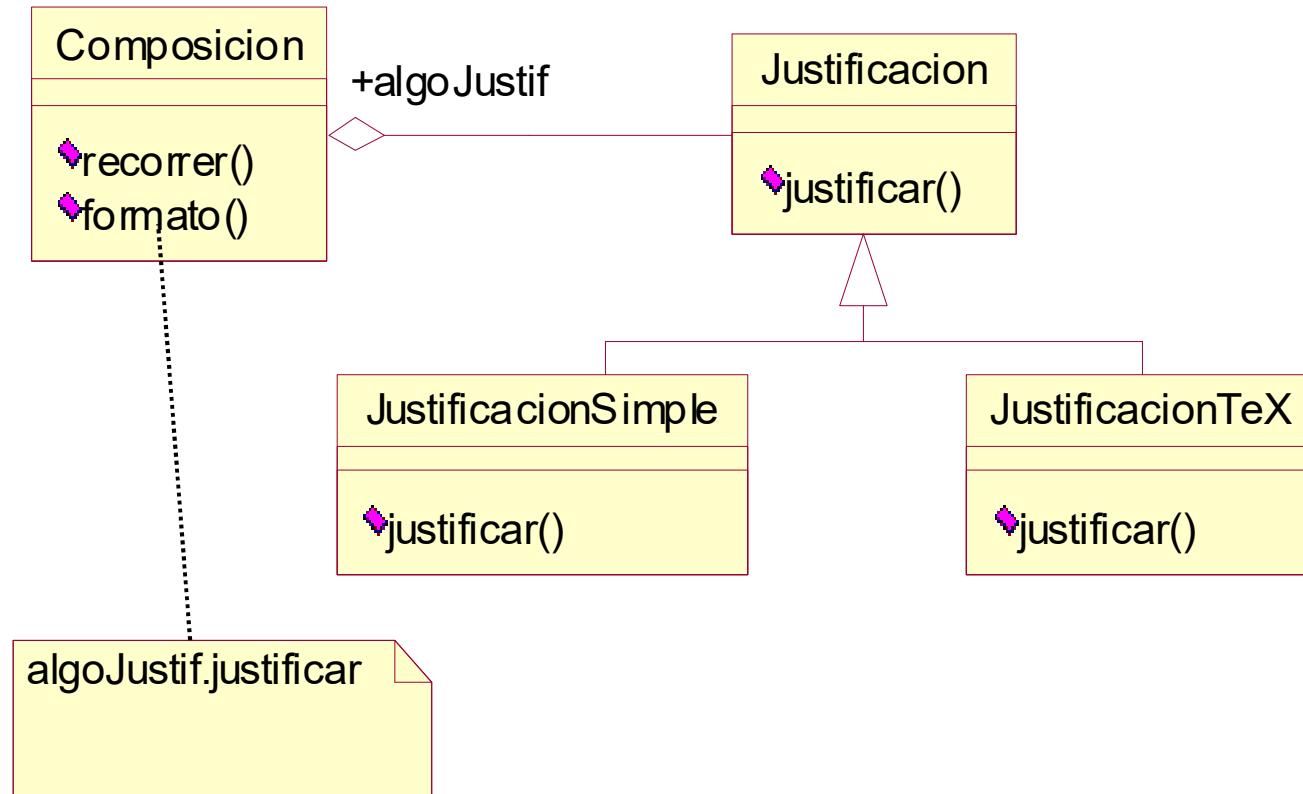
- ***Observando el Diseño general resultante:***
- En lugar de pensar en los comportamientos del Pato como un conjunto de comportamientos, podemos pensarlos como una *familia de algoritmos*.
- Los algoritmos representan cosas que los patos podrían hacer (diferentes maneras de quackear o volar).
- De igual manera se podría usar la misma técnica para un conjunto de clases que implemente las formas de computar tasas en diferentes provincias.
- Lo que se ha hecho es aplicar un Patrón de diseño: El Patrón Strategy

Strategy. Propósito y Motivación

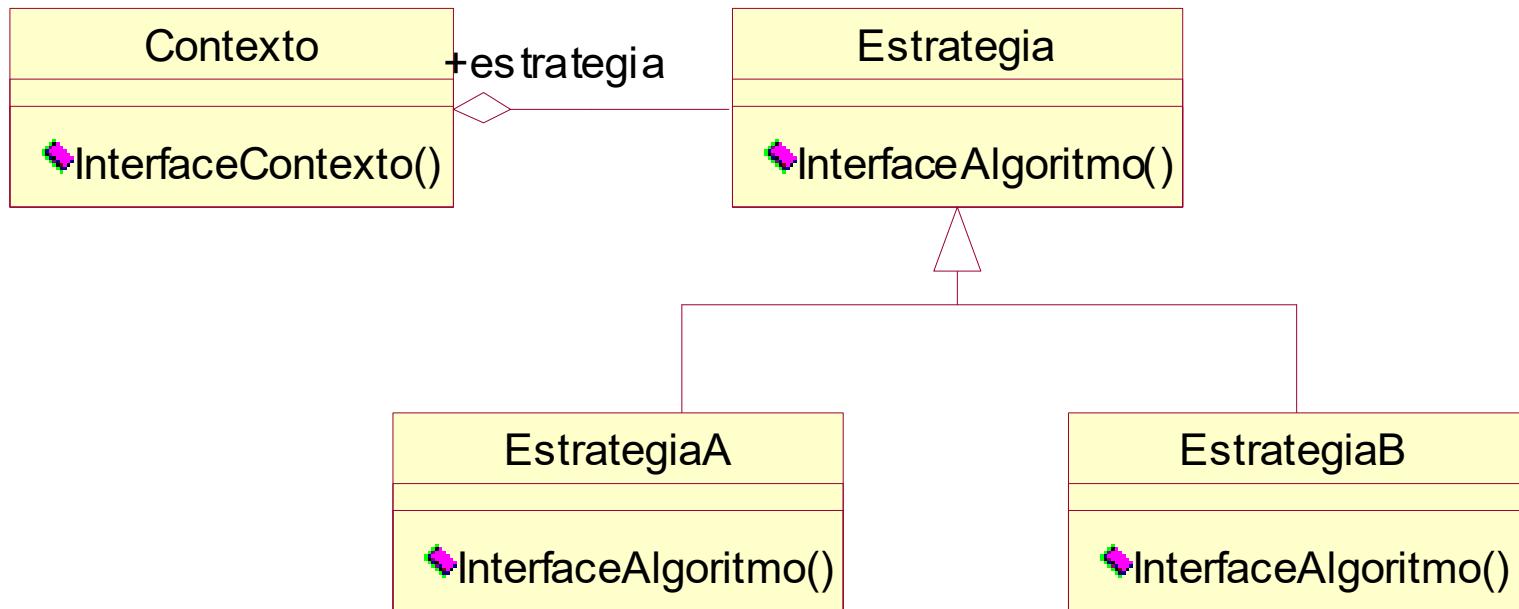
- **Propósito**
 - Define una **familia de algoritmos, encapsula cada uno, y permite intercambiarlos**. Permite variar los algoritmos de forma independiente a los clientes que los usan
- **Motivación**
 - Existen muchos algoritmos para justificación de texto, ¿debe implementarlo el cliente que lo necesita?

Strategy. Motivación

Motivación



Strategy. Estructura



Estructura

Strategy. Participantes

- **Estrategia:** Declara una interface común para todos los algoritmos que soporta. Context usa esta interface para llamar los algoritmos definidos con una estrategia concreta.
- **EstrategiaConcreta:** Implementa un algoritmo utilizando la interface Estrategia.
- **Contexto:**
 - Se configura con una estrategia concreta.
 - Mantiene una referencia al objeto Estrategia

Strategy. Participantes

- Estrategia y Contexto interactúan para implementar el algoritmo seleccionado. Un Contexto puede pasar todos los datos requeridos por el algoritmo a Estrategia cuando se llama el algoritmo. Alternativamente el contexto puede pasarse a sí mismo como argumento a las operaciones de Estrategia.
- Un contexto reenvía pedidos de sus clientes a sus estrategias. Los clientes usualmente crean y pasan un objeto EstrategiaConcreta al contexto exclusivamente.
- A menudo existe una familia de EstrategiaConcreta para que el cliente elija entre ellas.

Strategy. Consecuencias

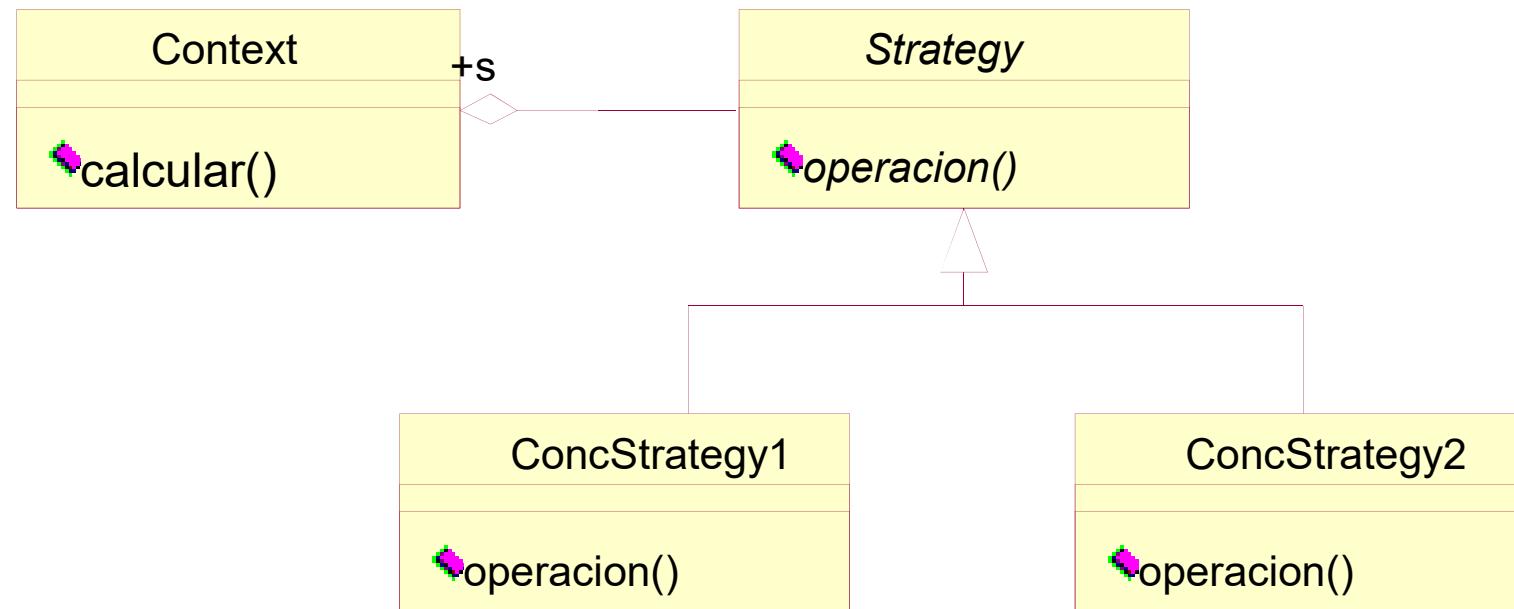
- **Consecuencias**
 - Define una familia de algoritmos relacionados.
 - Una alternativa a crear subclases de la clase *Contexto*.
 - Elimina sentencias CASE
 - El cliente puede elegir entre diferentes estrategias o implementaciones: debe conocer detalles
 - *State* y *Strategy* son similares, cambia el *Propósito*: ejemplos de composición con delegación

Strategy. Implementación

- **Implementación**

- ¿Cómo una estrategia concreta accede a los datos del contexto?
 - Pasar datos como argumentos
 - Pasar el contexto como argumento
 - *Estrategia* almacena una referencia al contexto

Strategy. Ejemplo



Strategy. Ejemplo

```
class Context {  
    Strategy s;  
    // PARA SELECCIONAR EL ALGORITMO  
    // SE ASIGNA A s LA INSTANCIA  
    // DE LA SUBCLASE QUE LO IMPLEMENTA  
    Context(Strategy str) {s = str;}  
    s.operacion(); }
```

Uso de Context desde un Cliente

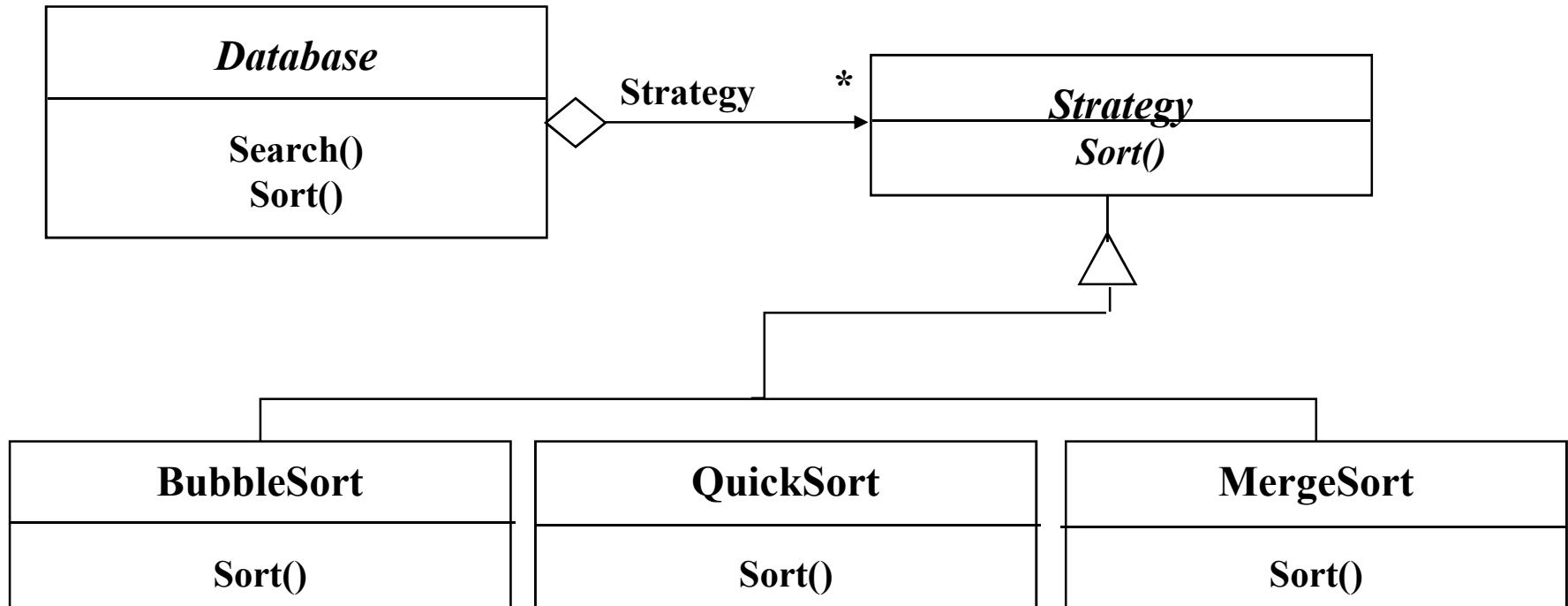
Context c = new Cliente(new ConcStrategy1());
Se indica cuál es el algoritmo que se ejecutará

Strategy. Ejemplo

```
class Context {  
    // EN VEZ DE:  
    if (cond1) operac1();  
    else if (cond2) operac2();  
    else ...  
}
```

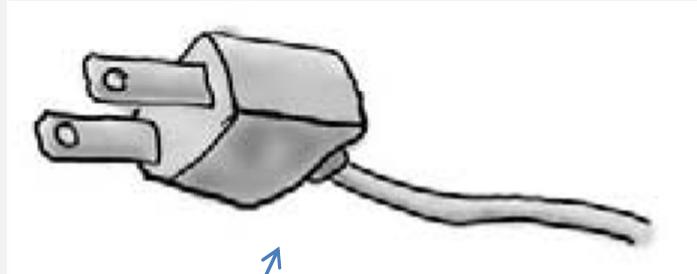
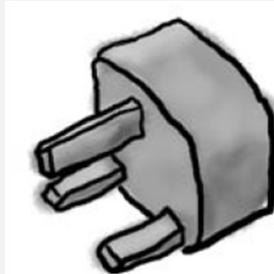
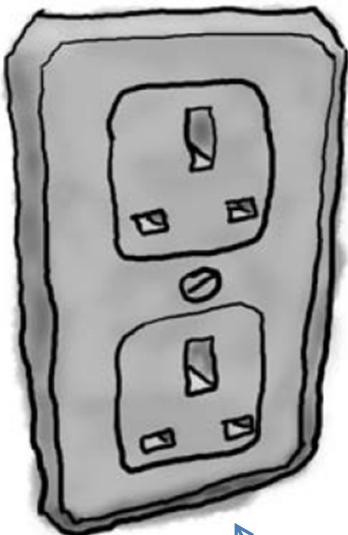
- Sentencias condicionales para seleccionar el algoritmo
- Si se quisiera añadir una nueva forma de ejecutar operacion() entonces HABRÍA QUE CAMBIAR EL CÓDIGO DE LA CLASE Context

Strategy en una Aplicación de Base de Datos



Adapter. Propósito

- **Propósito**



El **Adaptador** convierte una interface en otra

Los adaptadores OO hacen lo mismo: toman una interfaz y la adaptan a lo que un Cliente espera

Adapter. Ejemplo

- Definimos una interface Duck y su imp. MallardDuck

```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```

Esta vez, Duck implementa una interface que permite a los patos quackear y volar

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
    public void fly() {  
        System.out.println("I'm flying");  
    }  
}
```



Implementaciones simples

Adapter. Ejemplo

- **Contamos además con una jerarquía de Pavos (una interface y su clase de implementación)**

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

No hacen quack sino gobble

Pueden volar sólo pequeñas distancias

```
public class WildTurkey implements Turkey {  
    public void gobble() {  
        System.out.println("Gobble gobble");  
    }  
    public void fly() {  
        System.out.println("I'm flying a short distance");  
    }  
}
```

Adapter. Ejemplo

- **Nos faltan patos!! Queremos usar pavos en su lugar**
- **Creamos un Adapter de Pavos:**

```
public class TurkeyAdapter implements Duck {  
    Turkey turkey;  
  
    public TurkeyAdapter(Turkey turkey) {  
        this.turkey = turkey;  
    }  
  
    public void quack() {  
        turkey.gobble();  
    }  
  
    public void fly() {  
        for(int i=0; i < 5; i++) {  
            turkey.fly();  
        }  
    }  
}
```

Primero se implementa la interface del tipo que se está adaptando. Es la interface que el cliente espera.

Luego, se toma una referencia del objeto que se está adaptando.

Se implementan todos los métodos en la interface; **la traducción de quack entre clases se hace llamando al método gobble**.

Los pavos vuelan distancias cortas. Para realizar el mapeo a Patos, llamamos el método fly 5 veces.

Adapter. Ejemplo

```
public class DuckTestDrive {  
    public static void main(String[] args) {  
        MallardDuck duck = new MallardDuck(); Se crean un Pato y un  
        WildTurkey turkey = new WildTurkey(); Pavo  
        Duck turkeyAdapter = new TurkeyAdapter(turkey);  
  
        System.out.println("The Turkey says...");  
        turkey.gobble(); ← Testing del Pavo  
        turkey.fly();  
  
        System.out.println("\nThe Duck says...");  
        testDuck(duck); ← Testing del Pato  
  
        System.out.println("\nThe TurkeyAdapter says...");  
        testDuck(turkeyAdapter); ← Testing : pasar un Pavo como Pato  
    }  
    static void testDuck(Duck duck) {  
        duck.quack();  
        duck.fly();  
    }  
}
```

Este método toma un pato y llama sus métodos quack y fly

Luego se adapta el Pavo en un Adapter

Adapter. Test

Ejecución
del Test

```
File Edit Window Help Don'tForgetToDuck
```

```
%java RemoteControlTest

The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
```

El **Pavo** gobbles y
vuela una
distancia corta

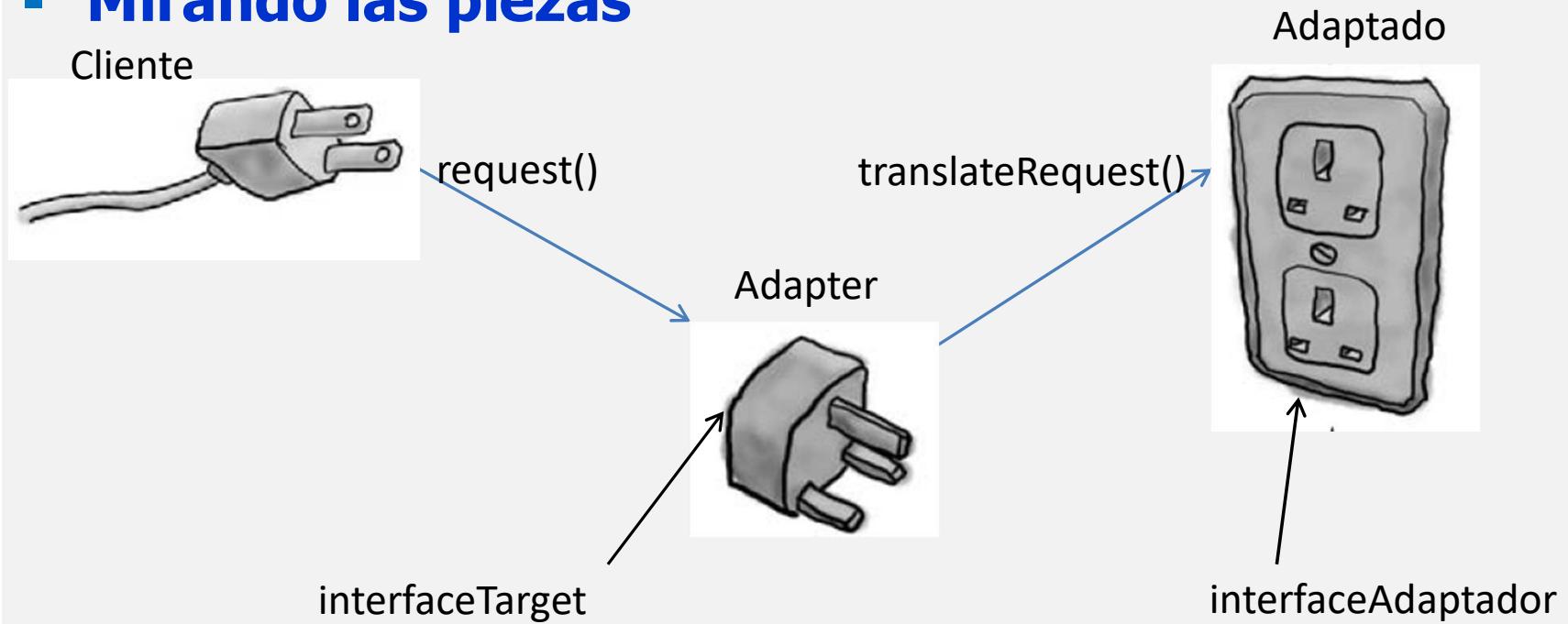
El **Pato** hace
quack y vuela
como se espera

El **Adapter** del Pavo
gobbles cuando
hace quack() y vuela
5 veces cuando se
llama fly()

El método testDuck() no se entera que tiene un pavo disfrazado de pato

Adapter explicado

■ Mirando las piezas



El **Adaptador** implementa la `interfaceTarget` y mantiene una instancia del **Adaptado**

Adapter explicado

- **Cómo usan el Adapter los Clientes**
- El Cliente hace un request al Adapter llamando un método en este usando la interfaceTarget
- El Adapter traduce el request en una o más llamadas al adaptado usando la interface del mismo.
- El Cliente recibe el resultado del llamado y nunca se entera que hay un adaptador haciendo la traducción.

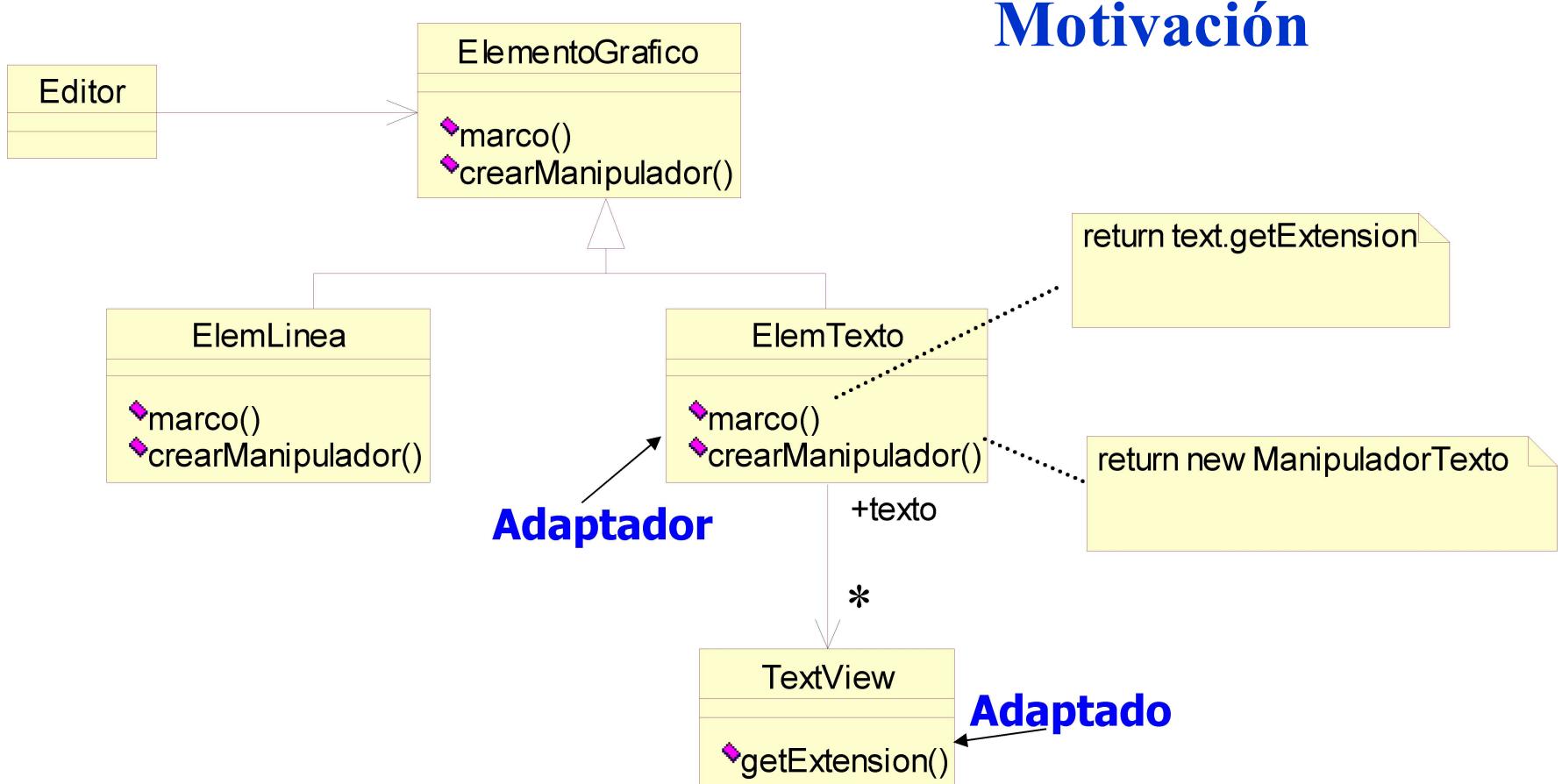
Adapter. Propósito y Motivación

- **Propósito**
 - **Convertir la interfaz de una clase en otra que el cliente espera.** Permite la colaboración de ciertas clases, trabajando juntas, a pesar de tener interfaces incompatibles
- **Motivación**
 - Un editor gráfico incluye una jerarquía que modela elementos gráficos (líneas, polígonos, texto,...) y se desea reutilizar una clase existente *TextView* para implementar la clase que representa elementos de texto, pero que tiene una interfaz incompatible.

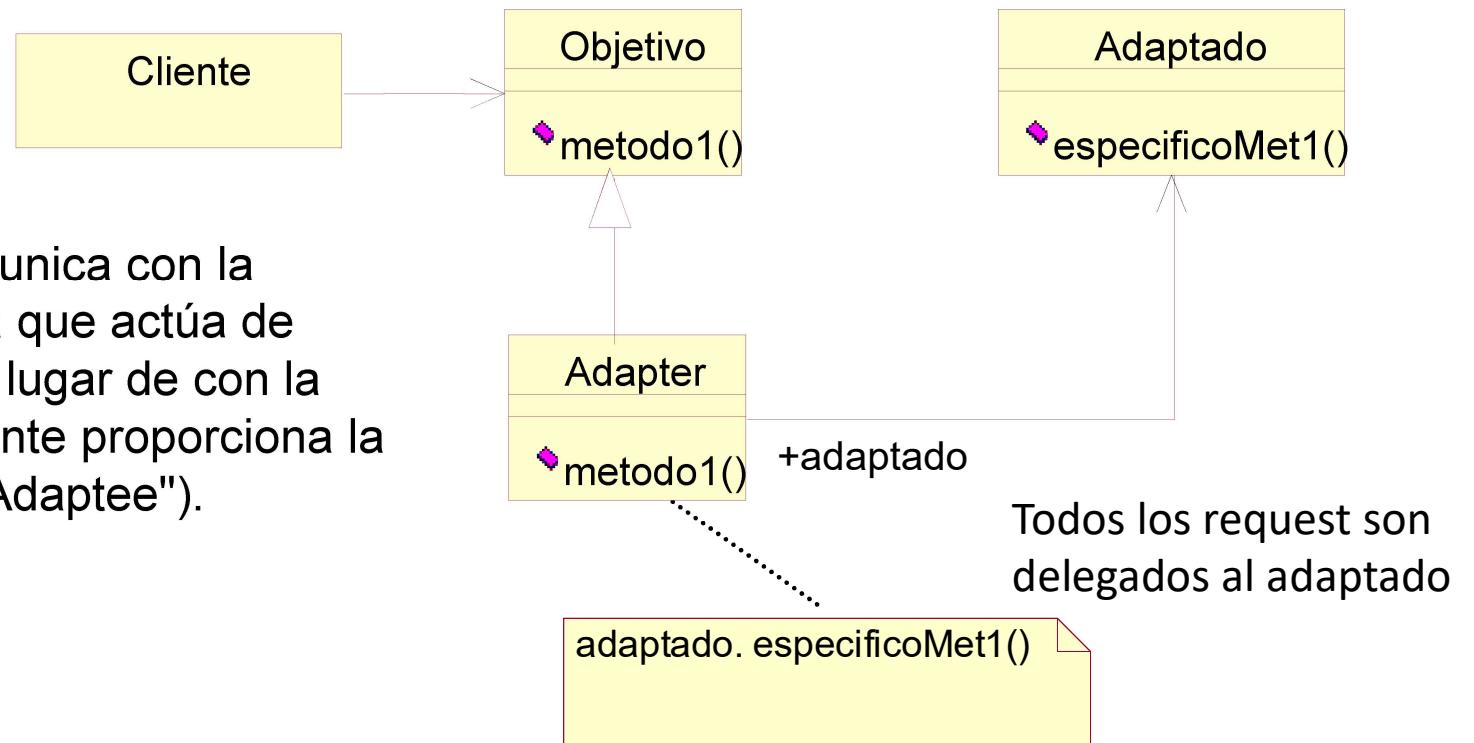
Adapter. Motivación

- **Motivación**
 - El problema suele presentarse cuando en una aplicación se reutilizan clases que no tienen exactamente la interfaz requerida.
 - La clase adaptadora se interpone entre el cliente y la clase reutilizada. La clase adaptadora presenta la interfaz que espera la aplicación, y se comunica con la clase reutilizada, "*traduciendo*" los mensajes.

Adapter. Motivación



Adapter. Estructura



Adapter. Aplicabilidad

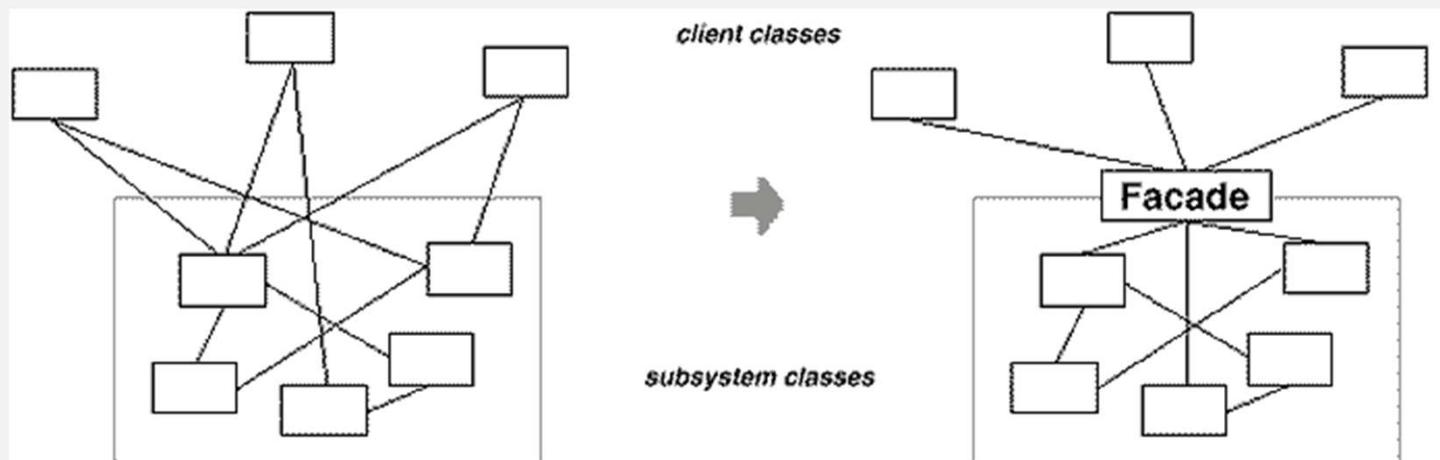
■ Aplicabilidad

- Se desea usar una clase existente y su interfaz no coincide con la que se necesita.
- Se desea crear una clase reutilizable que debe colaborar con clases no relacionadas o imprevistas

Facade. Propósito

■ Propósito

- Proporciona una interfaz unificada para un conjunto de interfaces de un sistema.
- Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar



Facade. Motivación

■ Motivación

- Estructurar un sistema en subsistemas ayuda a reducir la complejidad. Un típico objetivo de diseño es minimizar la comunicación y dependencias entre subsistemas.
- Un modo de lograr esto es introduciendo un objeto Fachada que proporcione una interfaz única y simplificada para los servicios más generales del subsistema.

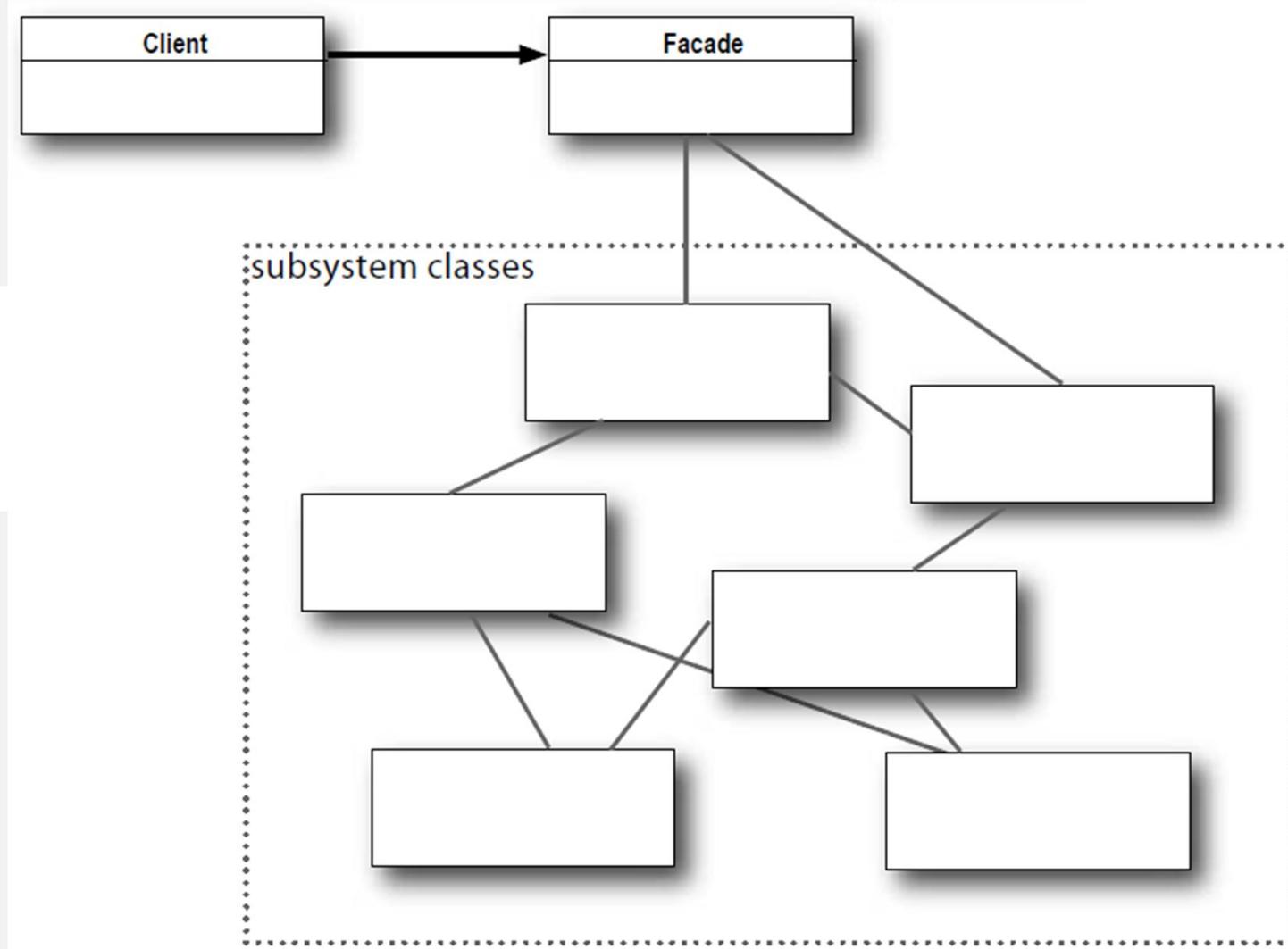
Facade. Aplicabilidad

■ **Aplicabilidad. Usamos Facade cuando**

- Queremos proporcionar una interfaz para un sistema complejo.
- Existe muchas dependencias entre los clientes y las clases que implementan una abstracción. Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas
- Se promueve la independencia entre subsistemas y la portabilidad
- Queremos dividir en capas nuestros subsistemas. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema

Facade. Estructura

■ Estructura



Principio de diseño vinculado

- **Principio de Menor Conocimiento: Hable sólo con sus amigos inmediatos**
- Nos guía a reducir las interacciones entre objetos sólo a los más cercanos
- Durante el diseño del sistema, se debe cuidar el número de clases con el cual se interactúa
- Este principio previene de crear diseños que tienen gran número de clases acopladas, de tal forma que un cambio en alguna parte del sistema provoca una reacción en cadena en otras.
- Cuando se generan muchas dependencias, se construyen sistemas frágiles difíciles de comprender y mantener

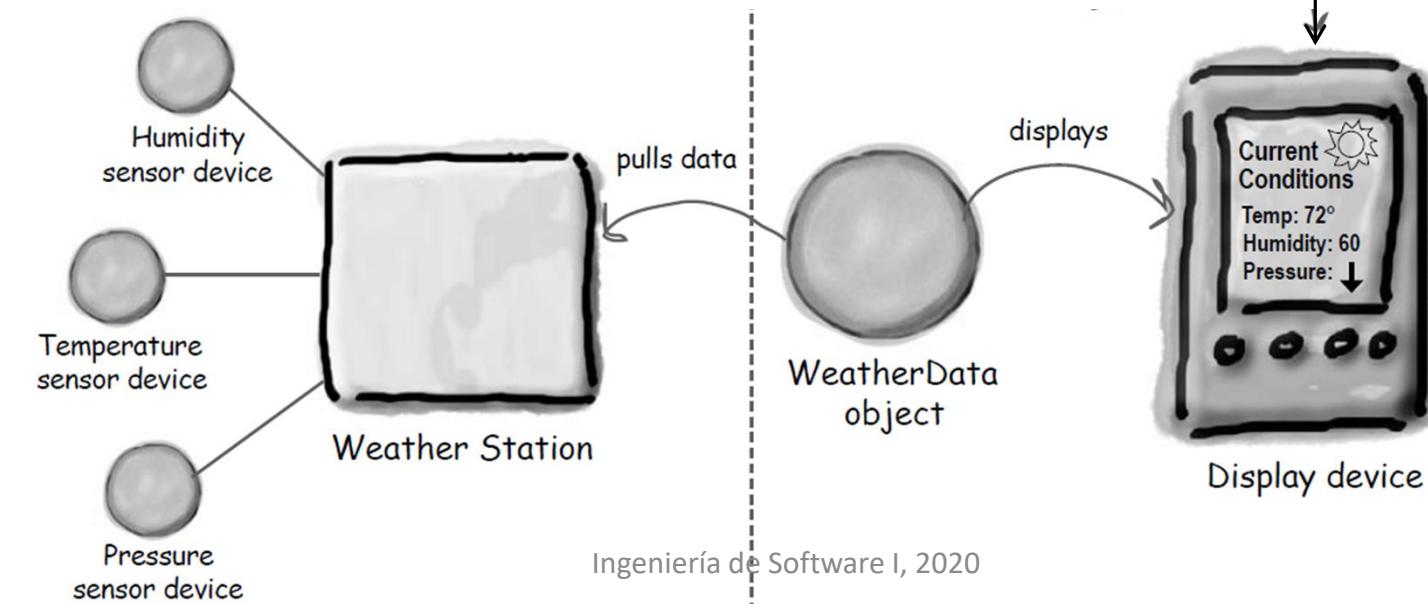
Patrón Observer. Introducción

- A menudo ocurre que un elemento de un diseño necesita tomar alguna acción cuando otro elemento en el diseño descubre que ocurrió un evento.
- Sin embargo, con frecuencia no queremos que el detector conozca acerca del actor.
- **Ejemplo:** El caso de una estación meteorológica que muestra el estatus de un sensor que captura datos.
 - ✓ Cada vez que el sensor cambia su lectura queremos que la estación muestre el nuevo valor.
 - ✓ Sin embargo, no queremos que el sensor conozca nada sobre la estación

Patrón Observer. Introducción

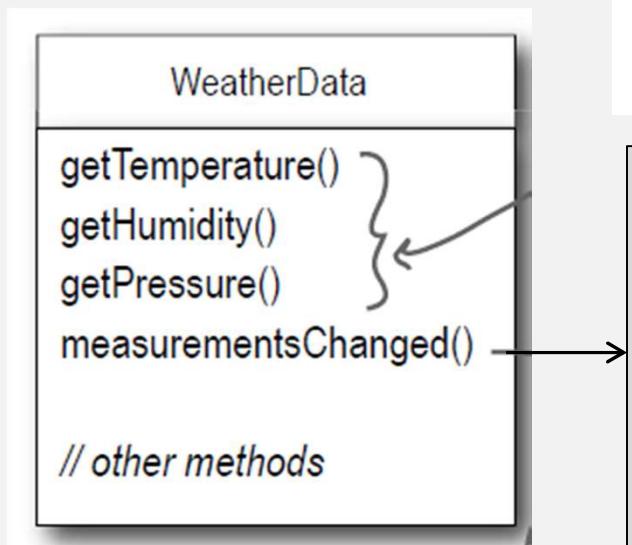
- Los 3 “jugadores” del sistema son: Estación meteorológica (Dispositivo físico que adquiere datos del mundo real), el Objeto WeatherData (que lleva los datos de clima provenientes de la Estación y actualiza el display) y el display que muestra a los usuarios las condiciones actuales del tiempo.

Condiciones actuales es uno de los 3 diferentes displays. El usuario puede ver estadísticas del tiempo y pronósticos



Patrón Observer. Introducción

- EL trabajo consistirá en implementar el método `measurementsChanged()`, de tal forma que actualice los 3 displays cuando se reciben nuevos datos.



Estos 3 métodos retornan las mediciones más recientes. Este objeto conoce cómo actualizarse desde la Estación Climática

```
/*
 * Este método se llama cada vez que se
 * actualizan las mediciones climáticas
 */
public void measurementsChanged) {
    // aquí va el código
}
```

Patrón Observer. Introducción

- Se debe crear una aplicación que use el objeto WeatherData para actualizar 3 displays cada vez que cambia la medición: display para condiciones actuales, estadísticas climáticas y pronósticos.
- El sistema debe ser expandible a nuevos displays



Future displays

Displays ofrecidos



Display One



Display Two



Display Three

Patrón Observer. Introducción

- Se muestra una posibilidad de implementación del método `measurementsChanged()`.

```
public class WeatherData {  
  
    // instance variable declarations  
  
    public void measurementsChanged() {  
  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
  
    // other WeatherData methods here  
}
```

Almacena las mediciones recientes
llamando los métodos getters (ya
implementados)

Actualiza el
display

Llama cada elemento del
display pasándole los valores
de mediciones recientes

Problema en la implementación del método

- Cuáles son los problemas en la implementación del método `measurementsChanged()` propuesto?

```
public void measurementsChanged() {  
  
    float temp = getTemperature();  
    float humidity = getHumidity();  
    float pressure = getPressure();  
  
    currentConditionsDisplay.update(temp, humidity, pressure);  
    statisticsDisplay.update(temp, humidity, pressure);  
    forecastDisplay.update(temp, humidity, pressure);  
}  
}
```

Area de cambios;
Se debe encapsular

Codificando a implementaciones concretas no hay forma de agregar o remover otros elementos display sin hacer cambios al programa

Se está usando una interface común para hablar con los elementos de display.
Todos tienen un método `update()` que toma los mismos parámetros

Patrón Observer. Publisher - Subscribers

- Cómo trabaja la suscripción y baja a diarios o revistas?
 1. Una editorial de periódicos entra en el negocio y comienza a publicar diarios
 2. Una persona se suscribe a un editor particular y cada vez que hay una nueva edición, se le envía. Mientras se mantiene como suscriptora, recibe nuevos diarios
 3. La persona se da de baja cuando no quiere más diarios, y la editorial deja de enviarlos
 4. Mientras el editor permanece en el negocio, personas, hoteles, y otros negocios permanentemente se suscriben y dan de baja al periódico

Publicador + Suscriptores = Patrón Observer



Se llama Subject al Publicador y Observers a los suscriptores

Entendiendo el Patrón Observer

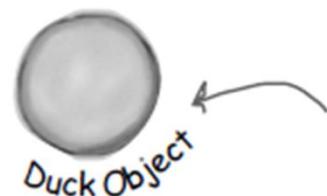


Cuando los datos en el Subject cambian, se notifica a los observers

El objeto Subject maneja algún dato

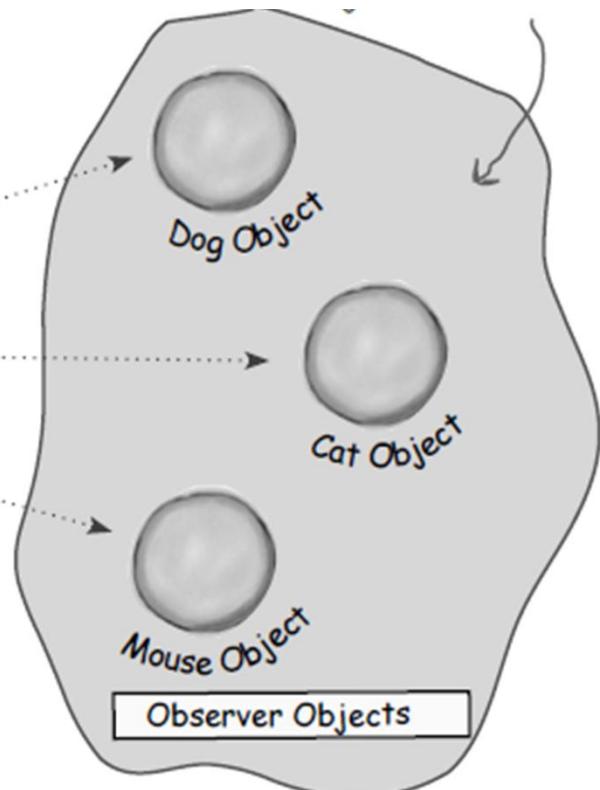


Los nuevos valores de datos son comunicados a los observers en alguna forma cuando cambian



Este objeto no es un Observer, de modo que no es notificado de cambios en el Subject

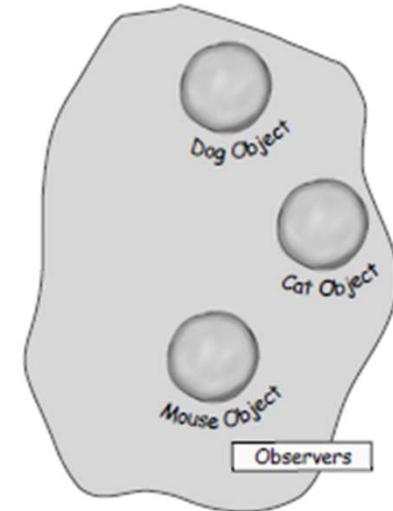
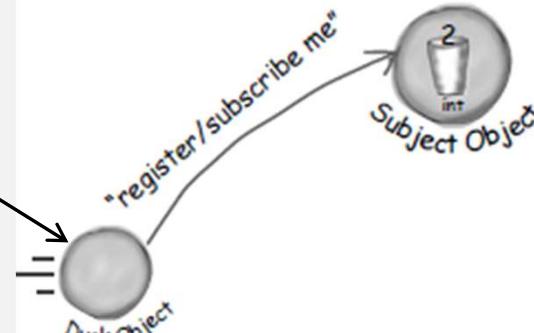
Los observers se han suscripto al Subject para recibir actualizaciones cuando cambian los datos del Subject



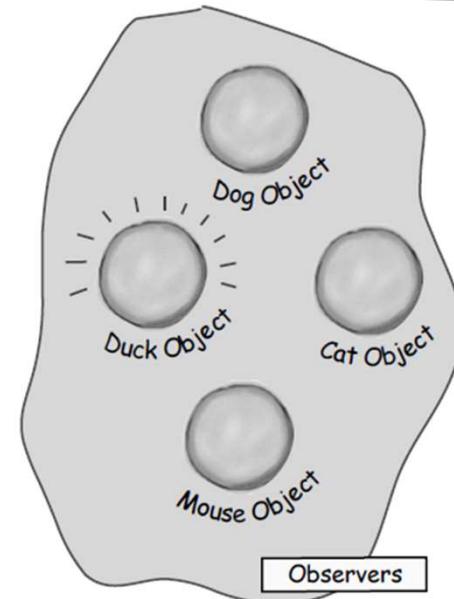
Entendiendo el Patrón Observer

■ Llega un nuevo Observer

Un objeto Duck llega y le avisa al Subject que quiere convertirse en Observer



El Objeto Duck es ahora un Observer oficial.
En la próxima notificación recibirá un entero

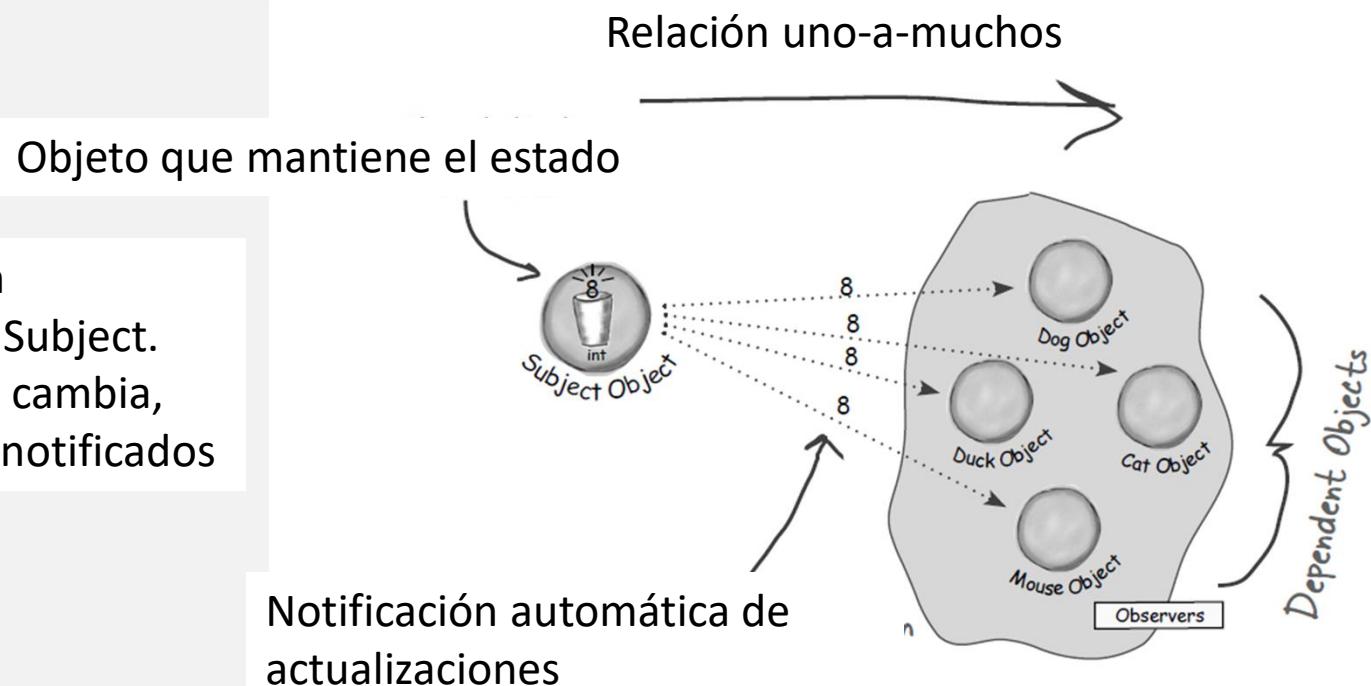


De igual forma, algún objeto, como Mouse, puede dejar de ser Observer

El Patrón Observer como solución

- Podemos manejar esta situación con un OBSERVER.
- El Patrón Observer define una dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia su estado, todas sus dependencias son notificadas y actualizadas automáticamente.

Los Observers son dependientes del Subject. Cuando su estado cambia, los observers son notificados

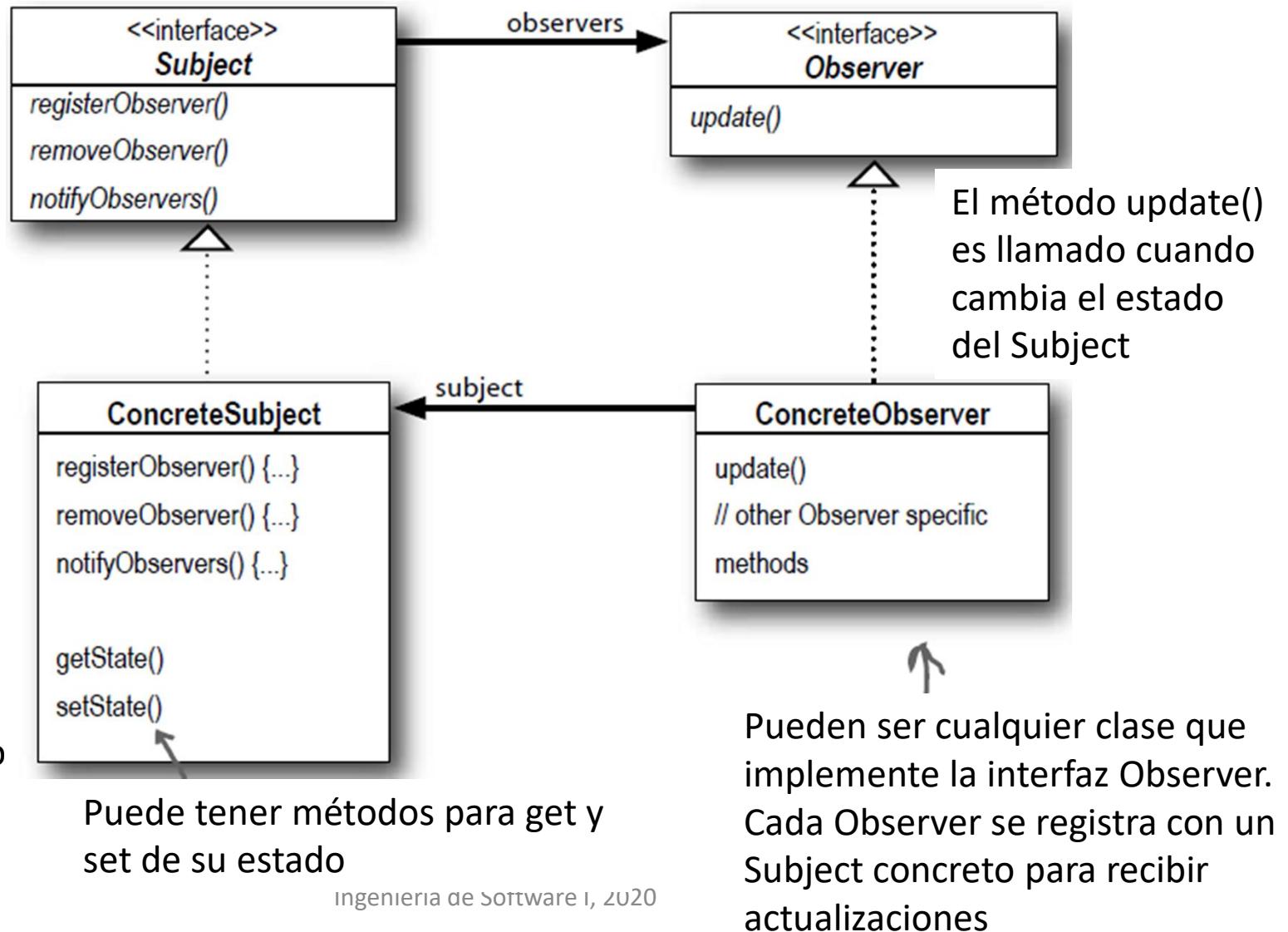


El Patrón Observer

Interface del Subject. Los objetos la usan para registrarse como Observers y para dejar de serlo

Implementa la interface Subject.
Implementa el método notifyObservers() que se usa para actualizar los obsevrers cuando cambia el estado

Cada Subject puede tener muchos observers



El Patrón Observer y bajo acoplamiento

- El Patrón Observer provee un diseño de Objetos donde subjects y observers están poco acoplados
 - ✓ Lo único que el Subject conoce de un Observer es que implementa una cierta interface, no necesita conocer la clase concreta de Observer o lo que hace
 - ✓ Se puede agregar nuevos Observers en cualquier momento. La única dependencia del Subject es con una lista de objetos que implementan la interface
 - ✓ No se necesita modificar el subject para agregar nuevos tipos de observers. Solo tenemos que implementar la interface Observer en una nueva clase y registrarla como un Observer
 - ✓ Pueden reusarse Subjects y observers en forma independiente unos de otros.
 - ✓ Cambios en el subject o en un observer no afectarán al otro.

El Patrón Observer y bajo acoplamiento

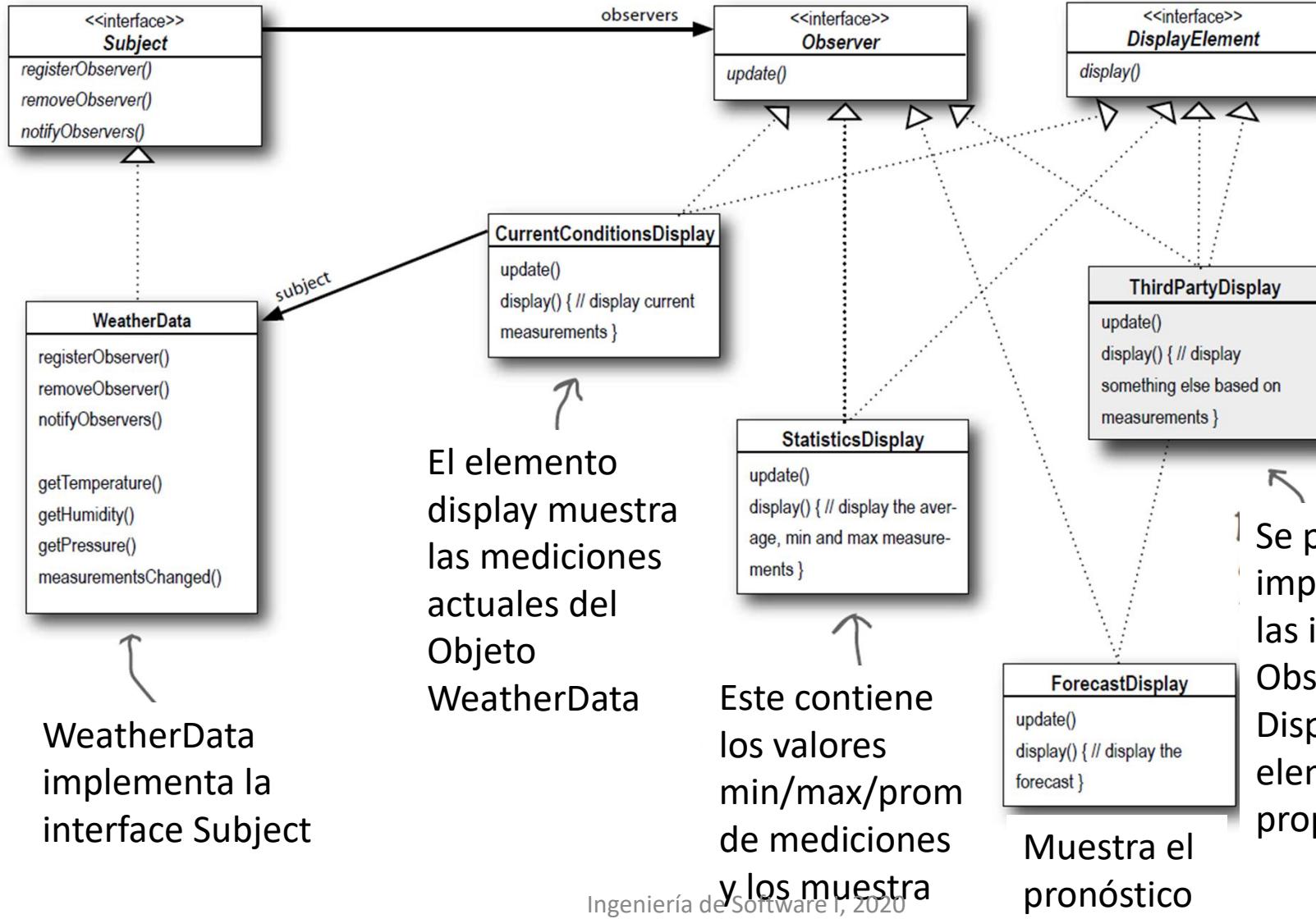
- *Principio de Diseño: Esforzarse en Diseños con bajo acoplamiento entre objetos que interactúan*
- Diseños con bajo acoplamiento permiten construir sistemas OO flexibles que pueden manejar cambios, dado que minimizan la interdependencia entre objetos.
- Cuando dos objetos están pobremente acoplados, pueden interactuar pero tienen poco conocimiento uno del otro.
- El Patrón Observer provee un diseño de Objetos donde subjects y observers están poco acoplados

Diseño Weather Station

Esta es la interface Subject

Le brinda al Subject una interface común para hablar cuando es tiempo de actualizar observers

Interface para implementar elementos de display



Implementando Weather Station

■ Implementamos las interfaces:

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}
```

Ambos métodos toman un Observer como arg., para registrar y remover

Método invocado para notificar a los Observers que el Subject cambió

```
public interface Observer {  
    public void update(float temp, float humidity, float pressure);  
}
```

Valores de estado que los observers toman del Subject cuando cambia la medición

Todos los observers tienen que implementar update()

```
public interface DisplayElement {  
    public void display();  
}
```

Implementando la Interface Subject en WeatherData

- Volvemos a implementar WeatherData con el Patrón Observer:

```
public class WeatherData implements Subject {  
    private ArrayList observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList();  
    }  
  
    public void registerObserver(Observer o) {  
        observers.add(o);  
    }  
  
    public void removeObserver(Observer o) {  
        int i = observers.indexOf(o);  
        if (i >= 0) {  
            observers.remove(i);  
        }  
    }  
  
    public void notifyObservers() {  
        for (int i = 0; i < observers.size(); i++) {  
            Observer observer = (Observer)observers.get(i);  
            observer.update(temperature, humidity, pressure);  
        }  
    }  
}
```

Here we implement the Subject Interface.

WeatherData implementa la interface Subject

Se agrega una lista para almacenar los observers y se crea en el constructor

Cuando el observer se registra se lo agrega al final de la lista

De igual manera, se lo saca de la lista

Aquí se les avisa a los observers sobre el estado. Como son observers, todos implementan update()

Implementando la Interface Subject en WeatherData

- Volvemos a implementar WeatherData con el Patrón Observer (Continuación):

```
public void setMeasurements(float temperature, float humidity, float pressure) {  
    this.temperature = temperature;  
    this.humidity = humidity;  
    this.pressure = pressure;  
    measurementsChanged();  
}  
  
// other WeatherData methods here  
}
```

Usaremos este método para hacer el test de los elementos display



Patrón Observer

■ Se construyen los elementos de display

El Display implementa Observer para que pueda tomar los cambios del objeto WeatherData

Implementa DisplayElement porque el API requerirá todos los elementos de display para implementar esta interface

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity; ←
        display(); ←
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

Se le pasa un objeto WeatherData (el subject) al constructor y se utiliza para registrar el display como un observer

Cuando se llama update(), se guardan la temp y hum y se llama display()

El método display() imprime los valores más recientes de las variables

■ Se realiza el test de Weather Station

```
public class WeatherStation {  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
  
        CurrentConditionsDisplay currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);  
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);  
  
        weatherData.setMeasurements(80, 65, 30.4f);  
        weatherData.setMeasurements(82, 70, 29.2f);  
        weatherData.setMeasurements(78, 90, 29.2f);  
    }  
}
```

Se simulan nuevas mediciones

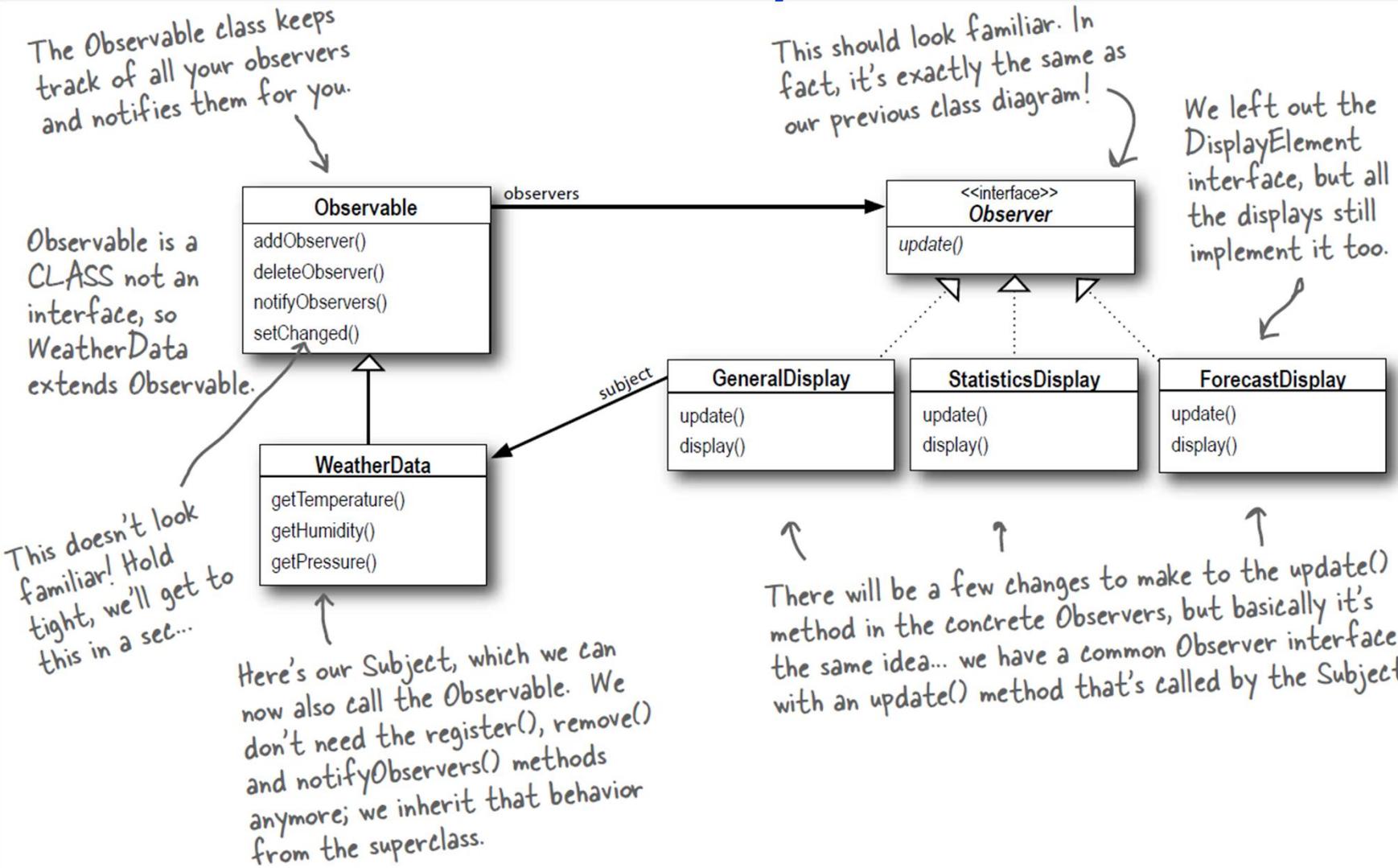
Primero se crea un objeto WeatherData

Se crean 3 displays y se les pasa el objeto WeatherData

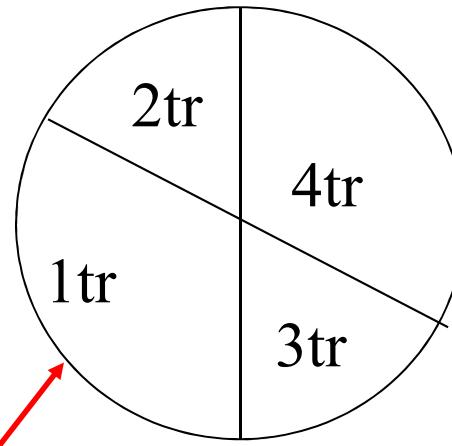
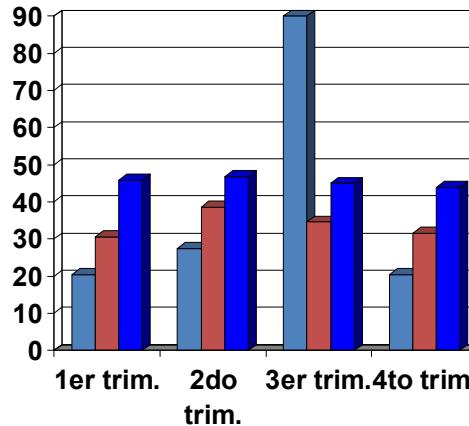
```
File Edit Window Help StormyWeather  
%java WeatherStation  
Current conditions: 80.0F degrees and 65.0% humidity  
Avg/Max/Min temperature = 80.0/80.0/80.0  
Forecast: Improving weather on the way!  
Current conditions: 82.0F degrees and 70.0% humidity  
Avg/Max/Min temperature = 81.0/82.0/80.0  
Forecast: Watch out for cooler, rainy weather  
Current conditions: 78.0F degrees and 90.0% humidity  
Avg/Max/Min temperature = 80.0/82.0/78.0  
Forecast: More of the same  
%
```

Soporte Built-in de Java para Observer

■ Usando la clase Observable y la Interface Observer

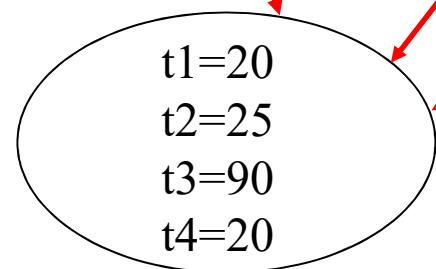


Patrón Observer



Vistas

Modelo



	<u>1tr.</u>	<u>2tr.</u>	<u>3tr.</u>	<u>4tr.</u>
Este	20	25	90	20
Oeste	30	38	32	32
Norte	47	47	45	45

Patrón Observer. Separación Modelo-Vista

■ Justificación

- Clases cohesivas
- Permitir separar el desarrollo de las clases de la vista y del dominio
- Minimizar el impacto de los cambios en la interfaz sobre las clases del modelo.
- Facilitar conectar otras vistas a una capa del dominio existente.
- Permitir varias vistas simultáneas sobre un mismo modelo.
- Permitir que la capa del modelo se ejecute de manera independiente a la capa de presentación.

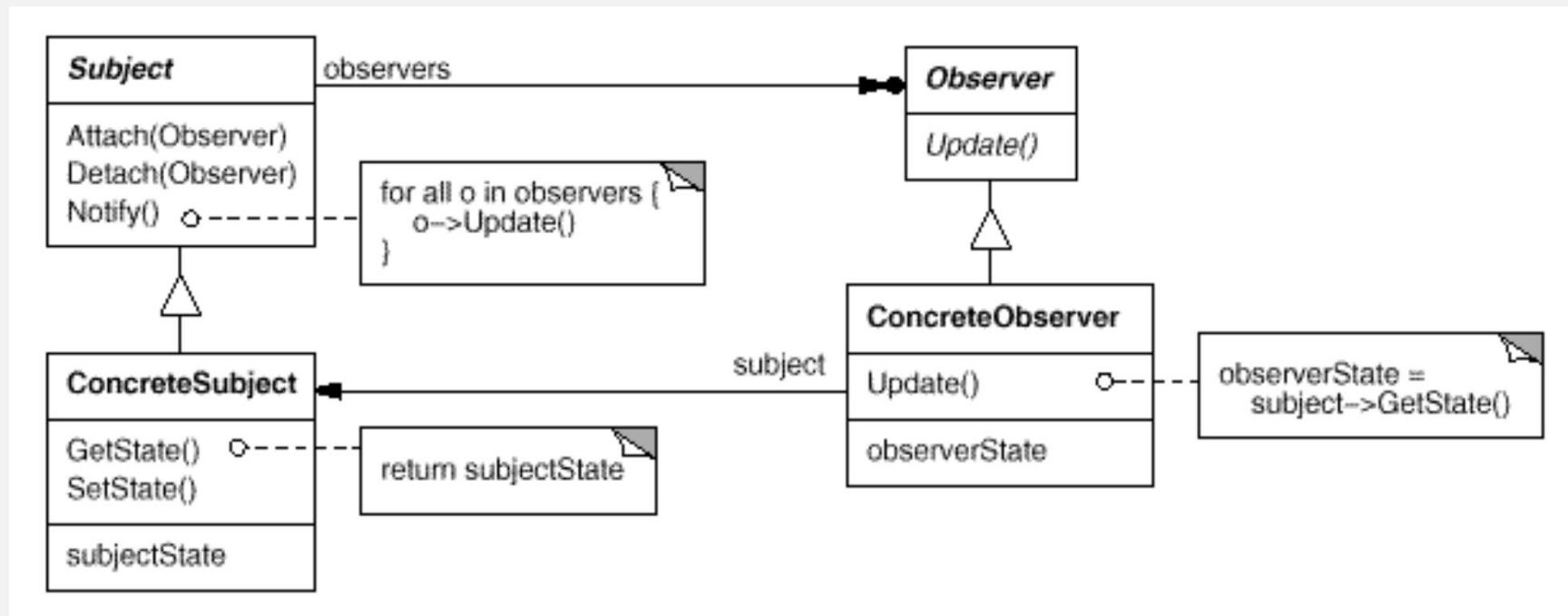
Patrón Observer. Separación Modelo-Vista

■ Aplicabilidad

- Cuando una abstracción tiene dos aspectos, uno dependiente de otro. Encapsular estos aspectos en objetos separados permite variar y reusarlos independientemente.
- Cuando un cambio de estado en un objeto requiere cambios en otros objetos, y no se sabe cuántos objetos necesitan ser cambiados.
- Cuando un objeto debe ser capaz de notificar algo a otros objetos, sin hacer suposiciones sobre quiénes son estos objetos.

Patrón Observer. Separación Modelo-Vista

■ Estructura



Patrón Observer

■ Participantes

- **Subject**
- Conoce sus observers. Cualquier número de observers puede observar un subject.
- Provee una interface para adjuntar y despegar objetos obsevers.
- **Observer**
 - Define una interface de actualización para objetos que deben ser notificados de cambios en un subject.
- **ConcreteSubject**
 - Almacena el estado de interés para objetos ConcreteObserver.
 - Envía una notificación a sus observers cuando su estado cambia.
- **ConcreteObserver**
 - Mantiene una referencia a un objeto ConcreteSubject.
 - Almacena el estado que debe mantenerse consistente con el del subject.
 - Implementa el Observer actualizando su interface para mantener su estado consistente con el del subject.

Patrón Observer

■ Colaboraciones

- ConcreteSubject notifica a sus observers cada vez que ocurre un cambio que puede hacer que el estado de sus observers sea inconsistente con el suyo.
- Después de ser informado de un cambio en el subject concreto, un objeto ConcreteObserver puede consultar información al subject. ConcreteObserver usa esta información para hacer consistente su estado con el del subject.

Más Patrones GRASP

- Polimorfismo
- Indirección
- Fabricación pura
- Variaciones Protegidas

Patrón Polimorfismo

- **Polimorfismo**
- **Problema:** ¿Cómo manejar las alternativas basadas en el tipo? ¿De qué manera crear componentes de software conectables?
 - **Alternativas basadas en el tipo:** La variación condicional es un tema esencial en la programación. Si se diseña un programa mediante la lógica condicional if-then-else o case, habrá que modificar la lógica del case cuando surja una variante.
 - Este enfoque dificulta extender un programa con nuevas variantes, porque los cambios tienden a requerirse en varios lugares donde exista la lógica condicional.
 - **Componentes de Software conectables:** Viendo los componentes en las relaciones cliente - servidor, ¿cómo podemos remplazar un componente servidor con otro sin afectar al cliente?

Patrón Polimorfismo

- **Polimorfismo**
- **Solución:** Cuando las alternativas o comportamientos relacionados varían por el tipo, las responsabilidades del comportamiento se asignarán - mediante operaciones polimórficas - a los tipos en los que el comportamiento varía.
- Corolario: no realizar pruebas con el tipo de un objeto ni utilizar lógica condicional para plantear diversas alternativas basadas en el tipo.
- *Polimorfismo* significa “asignar el mismo nombre a servicios en varios objetos”, cuando los servicios se parecen o están relacionados entre sí

Patrón Polimorfismo

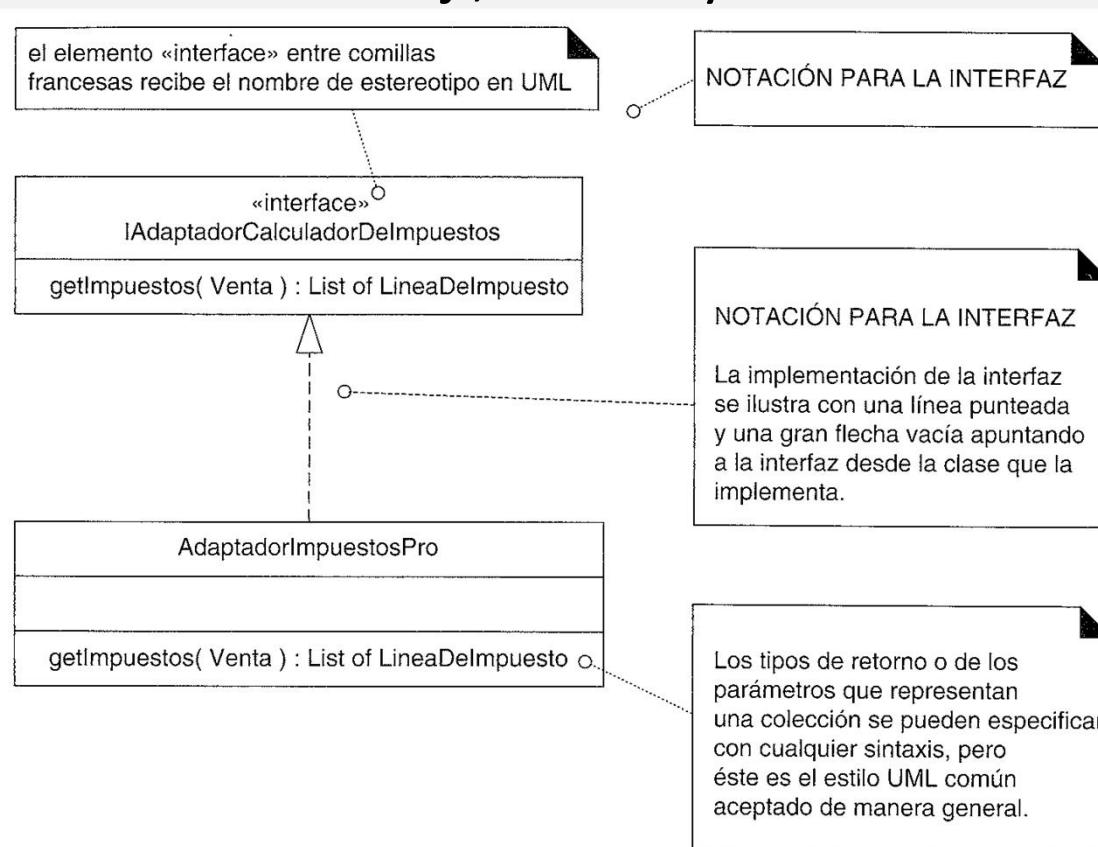
- El uso del patrón Polimorfismo está acorde al espíritu del patrón Experto.
- Si podemos caracterizar Experto como el patrón fundamental táctico, Polimorfismo será el más importante patrón estratégico en el diseño orientado a objetos
 - ✓ Es un principio fundamental en que se fundan las estrategias globales, o planes de ataque, al diseñar cómo organizar un sistema para que se encargue del trabajo.
- Beneficios: Es fácil agregar las futuras extensiones que requieren las variaciones imprevistas.

Ejemplo Adaptador Cálculo de Impuestos

- En el sistema PDV se deben soportar diferentes sistemas externos de cálculo de impuestos de terceras partes;
 - ✓ El sistema debe poder integrarse con distintos calculadores.
 - ✓ Cada calculador tiene un interfaz diferente; el comportamiento varía para adaptarse a cada uno de ellos
- Que objetos deberían ser responsables de manejar estas interfaces diferentes de los calculadores de impuestos externos?
- Puesto que el comportamiento para la adaptación del calculador varía según el tipo de calculador, según este Patrón se debería asignar la responsabilidad de la adaptación a distintos objetos adaptadores de calculador, implementada con una operación polimórfica *getImpuestos*

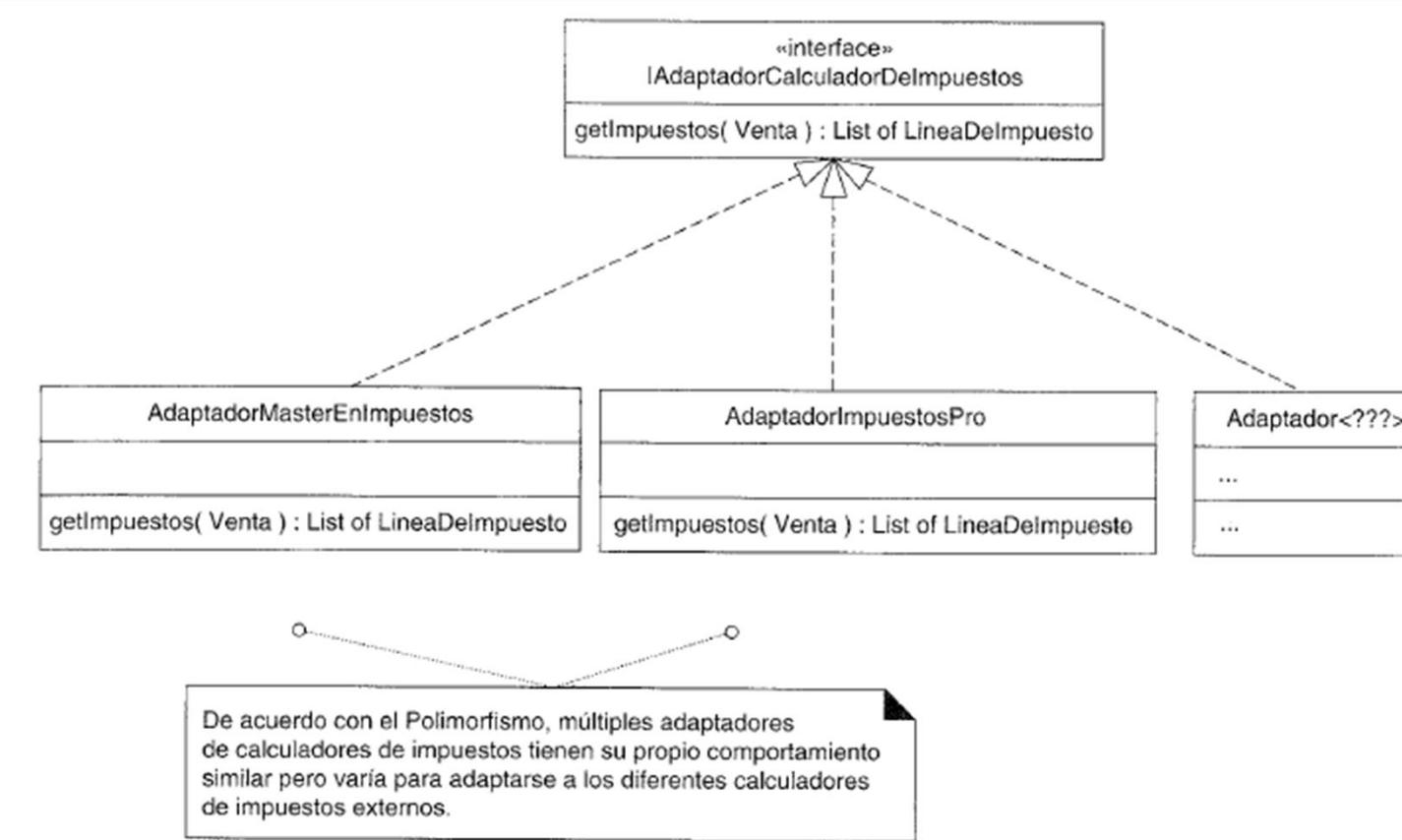
Ejemplo Adaptador Cálculo de Impuestos

- Usando Polimorfismo un diseño basado en asignación de responsabilidades puede extenderse fácilmente para manejar nuevas variaciones. Por ej., se incluye una nueva clase Adaptador.



Ejemplo Adaptador Cálculo de Impuestos

- Los adaptadores son objetos locales que representan los calculadores externos (quienes llamarán al calculador externo).



Patrón Fabricación Pura

- **Problema:** ¿A qué objeto se debería asignar una responsabilidad cuando no se quieren violar los patrones Alta Cohesión y Bajo Acoplamiento, pero la solución de aplicar el Experto no es adecuada?
 - ✓ Los diseños orientados a objetos se caracterizan por implementar como clases de software las representaciones de conceptos en el dominio de un problema del mundo real para reducir el gap semántico; por ejemplo, una clase **Venta**
 - ✓ Pese a ello, se dan muchas situaciones donde el asignar responsabilidades exclusivamente a las clases SW de la capa de dominio origina problemas de mala cohesión o acoplamiento o bien se obtiene un escaso potencial de reutilización

Patrón Fabricación Pura

- **Solución:** Asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial que no representa un concepto en el dominio del problema: una cosa inventada para dar soporte a: alta cohesión, bajo acoplamiento y reutilización.
 - ✓ Esa clase es una fabricación de la imaginación
 - ✓ Las responsabilidades asignadas a esa fabricación soportan alta cohesión y bajo acoplamiento (*diseño muy puro*).
 - ✓ Una fabricación pura implica *construir algo (Solución de emergencia!!!)*.

Patrón Fabricación Pura

- **Ejemplo** Supongamos, por ejemplo, que se necesita soporte para guardar las instancias **Venta** en una base de datos relacional.
- En virtud del patrón Experto, en cierto modo se justifica asignar esta responsabilidad a la clase **Venta**
- Pero no sería una buena idea por lo siguiente:
 - ✓ La tarea requiere un número relativamente amplio de operaciones de soporte orientadas a la base de datos, ninguna de las cuales se relaciona con el concepto de vender.
 - ✓ La clase **Venta** ha de ser acoplada a la interfaz de la base de datos relacional; de ahí que aumente su acoplamiento.
 - ✓ Guardar los objetos en una base de datos relacional es una tarea muy general en que debemos brindar soporte a muchas clases.

Patrón Fabricación Pura

■ Ejemplo

- En conclusión, aunque en virtud del patrón Experto es un candidato lógico para guardarse a sí misma en una base de datos, da origen a un diseño de baja cohesión, alto acoplamiento y bajo potencial de reutilización.
- Una solución razonable consiste en crear una clase nueva que se encargue tan sólo de guardar los objetos en algún tipo de almacenamiento persistente: una base de datos relacional.

Según la FabricaciónPura

Agente de Almacenamiento Persistente

guardar()

Patrón Fabricación Pura

- **Ejemplo**
- El AgenteAlmacenamientoPersistente no es un concepto del dominio (como Venta y Pago), sino algo fabricado para facilitar la programación.
- La Fabricación Pura soluciona los siguientes problemas de diseño:
 - ✓ La clase Venta permanece bien diseñada con alta cohesión y bajo acoplamiento
 - ✓ La nueva clase es cohesiva, dado que su único objetivo es la gestión de la persistencia.
 - ✓ La nueva clase es un objeto muy genérico y reusable.
- Las responsabilidades se cambiaron de la clase Venta a una Fabricación Pura.

Patrón Fabricación Pura

- **Explicación:** Para diseñar una Fabricación Pura debe buscarse ante todo un gran potencial de reutilización, asegurándose para ello de que sus responsabilidades sean pequeñas y cohesivas.
- Estas clases se crean para agrupar algún comportamiento común y, por la tanto se inspiran en una descomposición de comportamiento, en lugar de en una descomposición de representación.
 - ✓ Una fabricación pura suele partirse atendiendo a su funcionalidad y, por lo mismo, es una especie de objeto de función central.
 - ✓ Generalmente se considera que la fabricación es parte de la capa de servicios de alto nivel en una arquitectura orientada a objetos .

Patrón Fabricación Pura

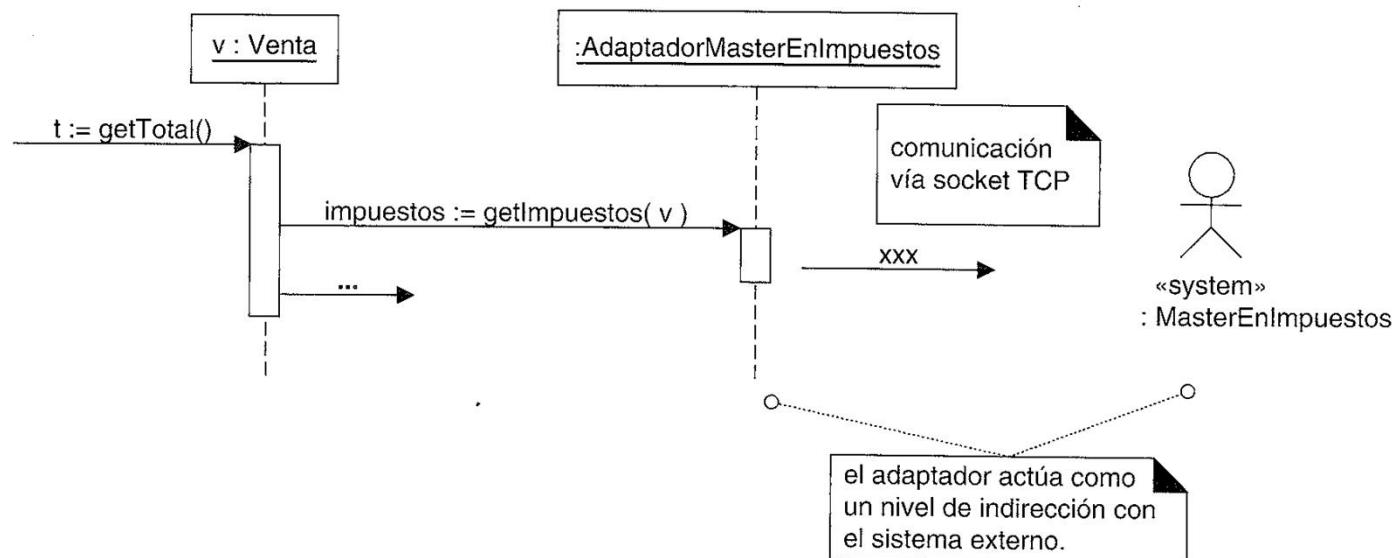
- **Problemas del excesivo uso:**
- Puede perderse el espíritu de los buenos diseños orientados a objetos que se centran en los posibles objetos y no en las funciones.
 - ✓ Las clases de Fabricación Pura casi siempre se dividen atendiendo a su funcionalidad; dicho con otras palabras, se confeccionan clases destinadas a conjuntos de funciones.
 - ✓ Si se abusa de ello, la creación de clases de Fabricación Pura, originará un diseño centrado en procesos o funciones que se implementa en un lenguaje orientado a objetos.

Patrón Indirección

- **Problema:** ¿A quién se asignarán las responsabilidades a fin de evitar el acoplamiento directo entre 2 o más clases?
 - ✓ ¿De qué manera se desacoplarán los objetos de modo que se obtenga un Bajo Acoplamiento y se conserve más alto el potencial de reutilización?
- **Solución:** Se asigna la responsabilidad a un objeto intermedio para que medie entre otros componentes o servicios, y éstos no terminen directamente acoplados.
 - ✓ El intermediario crea una indirección entre el resto de los componentes o servicios.

Patrón Indirección

- **Ejemplo:** Objeto que actúa como intermediario con los calculadores de impuestos externos
 - ✓ Mediante el polimorfismo proporcionan una interfaz para los objetos internos y ocultan variaciones en las APIs externas
 - ✓ Añadiendo un nivel de indirección y el polimorfismo, los objetos Adaptador protegen el diseño interno frente a las variaciones de las interfaces externas



Patrón Indirección

- **Ejemplo:** Agente de Almacenamiento Persistente
- El ejemplo de Fabricación Pura para desacoplar la Venta y los servicios de la base de datos relacional introduciendo la clase Agente de Almacenamiento Persistente es también un ejemplo de asignar responsabilidades para apoyar la Indirección.
- El Agente de Almacenamiento Persistente sirve de intermediario entre la Venta y la base de datos.

Patrón Variaciones Protegidas

■ Problema

- Cómo diseñar objetos, subsistemas y sistemas de modo que las variaciones o inestabilidad de esos elementos no tengan un impacto no deseado sobre otros elementos?

■ Solución

- Identificar puntos de variaciones predecibles o inestabilidad; asignar responsabilidades para crear una interface estable alrededor de ellos
- El término “interface” se usa en el sentido amplio de una vista de acceso, no sólo en el de interface Java.

Patrón Variaciones Protegidas

- **Ejemplo:** El problema del calculador de impuestos externo y su solución con el Polimorfismo
 - El punto de inestabilidad o variación lo forman las diferentes interfaces o APIs de los calculadores de impuestos externos.
 - El Sistema de PDV necesita poder integrarse con muchos sistemas de cálculo de impuestos existentes y también con futuros calculadores que todavía no existen
 - Añadiendo un nivel de indirección, una interfaz, y utilizando el polimorfismo con varias implementaciones de IAdaptadorCalculadorDelImpuestos, se protege el sistema de las variaciones en las APIs externas.
 - Los objetos internos colaboran con una interfaz estable; las distintas implementaciones del adaptador ocultan las variaciones de los sistemas externos.

Patrón Variaciones Protegidas

- *VP se aplica tanto a puntos de variación como de evolución*
- Un **punto de variación** representa a una variación del sistema actual contemplada en la especificación de requisitos o documento de entrada del diseño.
 - ✓ Por ejemplo: “el formato de compresión podrá ser PCX, GIF, BMP, TIFF y JPEG”
- Un **punto de evolución** es un punto de variación sobre cuya existencia se conjectura (especula) que podría existir en el futuro, pero que no está presente en los requisitos actuales .
 - ✓ Por ejemplo, a partir del requisito anterior, el diseñador puede especular sobre la evolución del sistema y tomar la decisión de protegerse sobre la variación del formato de compresión para dar cabida en el futuro a nuevos formatos (p.e a HSI-JPEG).

Bibliografía

- Design Patterns: Elements of Reusable Object-Oriented Software.
Gamma E, Helm R. Johnson R. y Vlissides J. Addison Wesley, 1995-2005
- Head First Design Patterns. Freeman E & E. Ed. O'Reilly, 2004
- UML y Patrones 3^a ed, Larman, 2005