PROG-3

UD 10

PRINCIPIOS DE DISEÑO ORIENTADO A OBJETOS

Pedro J. Ponce de León

Versión 20141210

Depto. Lenguajes y Sistemas Informáticos

Universitat d'Alacant
Universidad de Alicante

Indice

Introducción. Diseño orientado a objetos

- Diseño de software

  - DISEÑO: En el contexto del software, el diseño es un proceso de solución de problemas cuyo objetivo es encontrar y describir una forma de implementar los requerimientos funcionales de un sistema, respetando las restricciones impuestas por los principios de diseño, la plataforma y los requisitos del proceso tales como el presupuesto y los plazos.

Introducción. Diseño orientado a objetos

- **Diseño orientado a objetos**
  - Forma de diseñar donde se aplican, además de los principios de diseño generales del software, los principios específicos de diseño de software orientado a objetos, utilizando una plataforma de desarrollo (lenguaje, frameworks, librerías, etc.) orientada a objetos.
  - Los principios de diseño (generales u orientados a objetos) tienen como objetivo mejorar la calidad, tanto intrínseca como extrínseca del software (ver tema 1).
  - Damos por hecho: encapsulación, polimorfismo (enlace dinámico), herencia.

Principios generales de diseño

REPASO: **Acoplamiento y cohesión (de UD3)**

- **Acoplamiento**: relación entre componentes software
  - Interesa bajo acoplamiento. Éste se consigue moviendo las tareas a quién ya tiene habilidad para realizarlas.

- **Cohesión**: grado en que las responsabilidades de un solo componente forman una unidad significativa.
  - Interesa alta cohesión. Ésta se consigue asociando a un solo componente responsabilidades que están relacionadas en cierta manera, p. ej., acceso a los mismos datos.

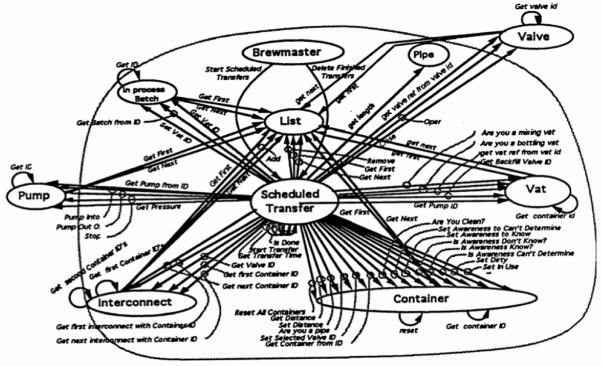
Principios generales de diseño

REPASO (UD3)

- Uno de los principios básicos de la ingeniería del software es "**incrementar la cohesión, reducir el acoplamiento**".

- Componentes con bajo acoplamiento y alta cohesión facilitan su utilización y su interconexión.

Principios generales de diseño

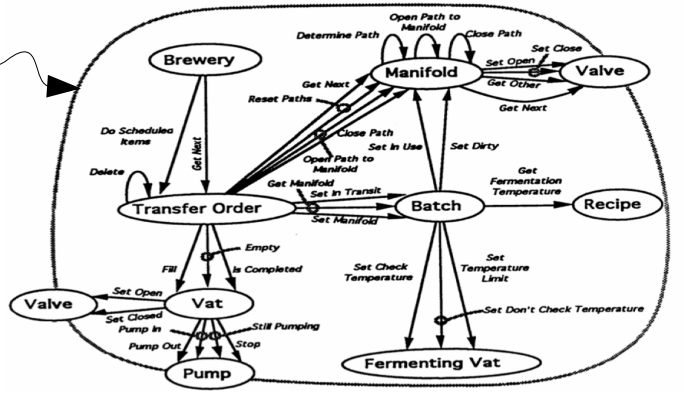


sistema demasiado acoplado

Acoplamiento vs. cohesión

Acoplamiento

sistema desacoplado y más cohesionado



Principios de diseño orientado a objetos

1. Principio abierto-cerrado
2. Principio de sustitución (Liskov)
3. Principio de responsabilidad única
4. Principio de segregación de interfaz
5. Principio de inversión de dependencias

1. Principio abierto-cerrado

- "Todos los sistemas cambian durante su ciclo de vida. Esto debe tenerse en cuenta al desarrollar sistemas que se pretende que duren más allá de la primera versión" (1)

- Principio abierto-cerrado
  - "***Las entidades software (clases, paquetes, métodos, etc.) deben estar abiertas a su extensión, pero cerradas a su modificación.***" (2)
  - Lo que ya funciona no se modifica (si acaso se refactoriza). Cuando los requisitos cambian, se extiende el comportamiento de estos módulos añadiendo código nuevo (nuevos métodos, clases, ...)

(1) Object Oriented Software Engineering a Use Case Driven Approach, Ivar Jacobson, Addison Wesley, 1992, p 21.

(2) Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, p 23

1. Principio abierto-cerrado

La herencia de interfaz sirve para esto:

- Las clases abstractas y sus derivadas son una forma de implementar conceptos fijos (cerrados) pero que al mismo tiempo representan un conjunto ilimitado de comportamientos.

- El código cliente que usa la interfaz de la clase abstracta puede cerrarse a la modificación, pero mantenerse abierto a la extensión mediante la creación de clases derivadas.

1. Principio abierto-cerrado

- Ejemplo

  Una aplicación debe ser capaz de dibujar círculos y cuadrados en un orden determinado. Dada una lista de círculos y cuadrados en el orden apropiado, la aplicación debe recorrerla y dibujar cada figura.

```
//Solución sin RTTI ni enlace dinámico

enum TipoFigura { circulo, cuadrado }
class Figura {
  private TipoFigura tipo;
  public getTipo() { return tipo; }
}
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujarCirculo()  { … }
}
class Cuadrado extends Figura {
  private Punto puntoSuperiorIzqda;
  private float lado;
  public void dibujarCuadrado()  { … }
}
```

```
// código cliente (¡apesta!)

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext()) {
    Figura fig = it.next();
    switch(fig.getTipo()) {
      case circulo:
        Circulo ci = (Circulo)fig;
        ci.dibujarCirculo(); break;
      case cuadrado:
        Cuadrado cu = (Cuadrado)fig;
        cu.dibujaCuadrado();  break;
    }
  }
}
```

1. Principio abierto-cerrado

- Ejemplo (cont.)
  - *dibujarFiguras()* no cumple el principio abierto-cerrado, porque no está cerrado a la adición de nuevas figuras.
  - Además, la sentencia switch aparecerá allá donde sea necesario discriminar entre diferentes tipos de figuras.

1. Principio abierto-cerrado

- Ejemplo (cont.)

```
// Solución con RTTI


class Figura { }
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujarCirculo()
{ … }
}
class Cuadrado extends Figura {
  private  Punto puntoSupIzqda;
  private float lado;
  public void dibujarCuadrado()
 { … }
}
```

```
// código cliente (¡sigue apestando!)

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext() {
    try {
      // downcasting no seguro:
      Circulo ci = (Circulo)it.next();
      Cuadrado cu = (Cuadrado)it.next();
      ci.dibujarCirculo();
      cu.dibujarCuadrado();
    } catch (ClassCastException cce) { … }
  }
}
```

dibujarFiguras() sigue sin cumplir el principio abierto-cerrado

(Sólo debemos usar RTTI cuando no viole el principio abierto-cerrado)

1. Principio abierto-cerrado

- Ejemplo (cont.)

  Solución que cumple el principio abierto-cerrado

```
abstract class Figura {
  abstract public void dibujar();
}
class Circulo extends Figura {
  private Punto centro;
  private float radio;
  public void dibujar()  { … }
}
class Cuadrado extends Figura {
  private Punto puntoSupIzqda;
  private float lado;
  public void dibujar()  { … }
}
```

```
// código cliente

void dibujarFiguras(List<Figura> lf) {
  Iterator<Figura> it = lf.iterator();
  while (it.hasNext() {
    Figura fig = it.next();
    fig.dibujar();
  }
}
```

dibujarFiguras() está cerrada frente a la adición de nuevas figuras.
Su comportamiento se puede extender sin cambiar su código, añadiendo nuevas clases derivadas de Figura.

1. Principio abierto-cerrado

- Una aplicación 'real' no puede estar 100% cerrada.
  - Por ejemplo, considera que sucedería si en el método dibujarFiguras() decidiéramos que los círculos deben ser dibujados antes que los cuadrados.
- Siempre existirá algún tipo de cambio para el cuál una aplicación no esté cerrada.
- El diseñador debe decidir, basándose en su experiencia, ante qué tipo de cambios va a cerrar su código.

1. Principio abierto-cerrado

- La clausura del código ante cambios se consigue mediante el uso de la abstracción.

- En el caso de nuestro ejemplo, lo que necesitaríamos sería alguna forma de abstraer la forma en que las figuras se ordenan: dadas dos figuras ¿cuál debemos dibujar primero? (*)

(*) La solución, aquí: http://www.objectmentor.com/resources/articles/ocp.pdf

1. Principio abierto-cerrado

- Heurísticas
  - Encapsulación:Todos los atributos deben tener visibilidad privada
    - Ningún método que dependa de un atributo puede ser cerrado con respecto a éste.
  - Variables globales no, gracias.
    - Ningún método que dependa de una variable global puede ser cerrado con respecto a ésta.
    - Existen ciertas excepciones, como `System.out`
  - Usar RTTI es peligroso (ejemplo pág. 18)

1. Principio abierto-cerrado

- **Conclusiones**
  - *Seguir este principio proporciona los mayores beneficios asociados a la tecnología orientada a objetos: reusabilidad y mantenibilidad.*
  - *Esto no se consigue simplemente usando un lenguaje orientado a objetos. Requiere aplicar la abstracción en aquellas partes de una aplicación que se prevee puedan cambiar.*

2. Principio de sustitución (Liskov)

Principio de sustitución (Liskov)

- Ya visto en UD 3, cuando hablamos de 'herencia de interfaz'.

- Enunciado original (de Barbara Liskov):
  - *“Si para cada objeto o1 de tipo S existe un objeto o2 de tipo T tal que, para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando se sustituye o2 por o1, entonces S es un subtipo de T"* - BarbaraLiskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988).

- “*El comportamiento de P no cambia*” no significa que su efecto es el mismo usemos un objeto de tipo S o T (S puede modificar el comportamiento base), sino que P no basa su interacción con los objetos o1 u o2 en saber si son de tipo S o T.

2. Principio de sustitución (Liskov)

- Enunciado alternativo:
  - Los métodos que usan referencias a clases base deben ser capaces de utilizar objetos de clases derivadas sin saberlo.

- Un método que no cumple este principio es aquél que, aún usando referencias a clase base, necesita conocer a las clases derivadas. No cumplirá tampoco el principio abierto-cerrado.

- Implica usar únicamente la interfaz de la clase base y respetar su contrato.

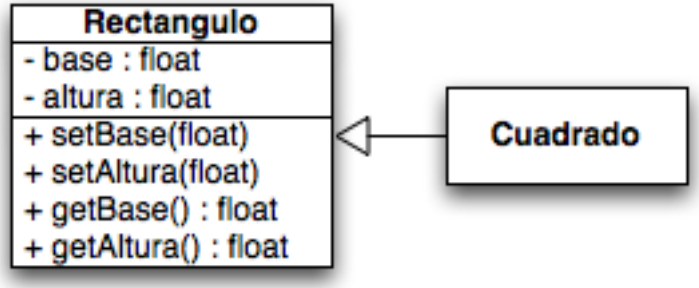
2. Principio de sustitución (Liskov)

Un ejemplo sutil...

Un cuadrado ES UN rectangulo cuya base es igual a su altura.

Pero,...

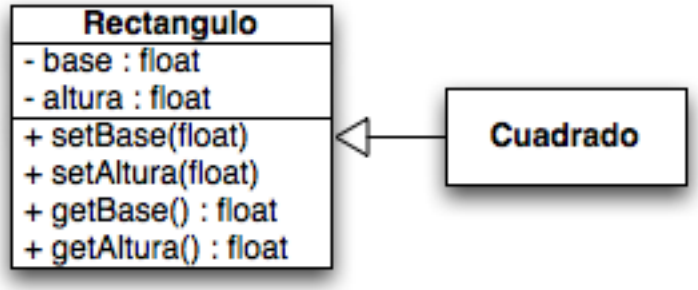
- En realidad no necesitamos dos atributos para el cuadrado.
- *setBase()* y *setAltura()* no son apropiadas para el cuadrado. Podemos sobreescribirlas.



```
void setBase(float b) {
   super.setBase(b);
   super.setAltura(b);
}

void setAltura(float a) {
   super.setBase(a);
   super.setAltura(a);
}
```

2. Principio de sustitución (Liskov)

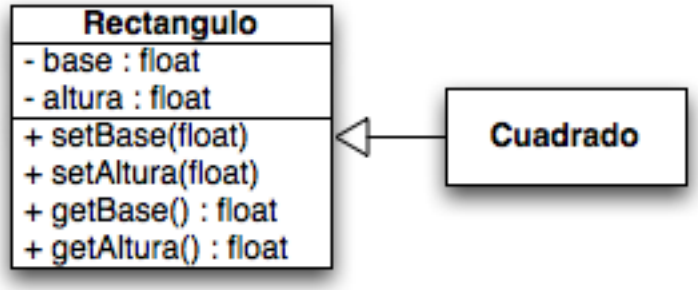


Ahora el modelo parece consistente, pero...

```
// codigo cliente: prueba unitaria
void testSetters(Rectangulo r) {
  r.setBase(5);
  r.setAltura(4);
  assertEquals("Setters", r.getBase() * r.getAltura(), 20);
}
```

- El problema: El programador del test asumió que cambiar la altura de un rectángulo no modifica su base.
- Por tanto, existen métodos que usan referencias a clase base, Rectangulo, que no funcionan correctamente con clases derivadas como Cuadrado.
- No se respetó el contrato: Un cuadrado no se comporta como un rectángulo. El principio de sustitución indica que en diseño OO, la relación ES-UN debe cumplirse en términos del comportamiento público (el contrato) y no sólo del estado interno.

2. Principio de sustitución (Liskov)



Diseño por contrato (1)

Ofrece mecanismos para asegurar que se cumple el principio de sustitución:

- Los métodos declaran precondiciones y postcondiciones:
  - Las precondiciones deben ser ciertas para que el método se pueda ejecutar
  - Cuando termina, el método garantiza que las postcondiciones se cumplen.

- Una posible postcondición para *Rectangulo.setAltura()* podría ser

```
assert( (altura == a) && (base == old.base) )
```

(1) *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

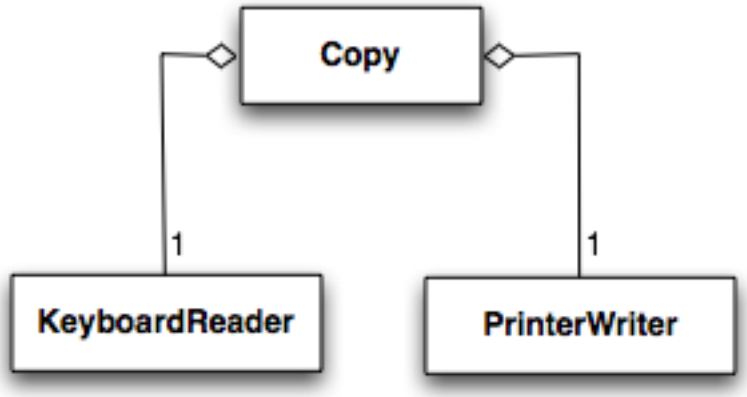
3. Principio de inversión de dependencias

- **Rasgos de un "mal diseño" en el software**
  - Es difícil de cambiar porque cualquier cambio afecta a demasiadas partes del sistema : **Rigidez**
  - Cuando realizamos un cambio, aparecen problemas en sitios inesperados: **Fragilidad**
  - Es complicado de reutilizar en otras aplicaciones, porque no puede ser desacoplado de la aplicación actual: **Inmovilidad**

- La interdependencia entre módulos es la culpable de un mal diseño.

3. Principio de inversión de dependencias

- **Ejemplo: El módulo 'Copiar'**




```
void Copy()
{
  int c;

  while ((c = KeyboardReader.read()) != EOF)
       PrinterWriter.write(c)
}
```

- Copy() no es reutilizable para copiar de algo que no sea un teclado a algo que no sea una impresora.

3. Principio de inversión de dependencias

- **Ejemplo: El módulo 'Copiar'**

Supongamos que deseamos poder copiar también de teclado a fichero:

```
enum OutputDevice {printer, disk}

void Copy(OutputDevice dev)
{
        int c;
        while ((c = KeyboardReader.read()) != EOF)
                if (dev == printer)
                        PrinterWriter.write(c);
                else DiskWriter.write(c);
}
```

Esto añade más dependencias. Cuanto más tipos de dispositivos añadimos, más dependencias aparecen.

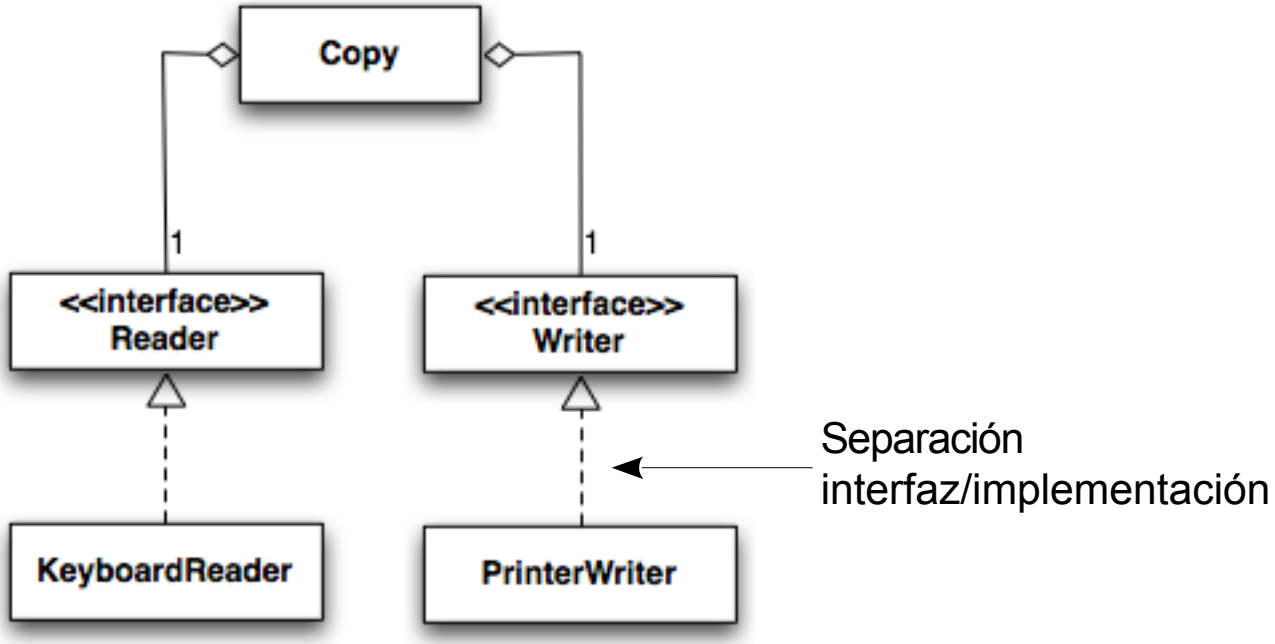
3. Principio de inversión de dependencias

- **Ejemplo: El módulo 'Copiar'**

Inversión de dependencias:

El módulo de alto nivel (Copy()), depende de los módulos de bajo nivel

Hemos de hacer Copy() independiente de los detalles de bajo nivel.



3. Principio de inversión de dependencias

- **Ejemplo: El módulo 'Copiar'**

```
interface Reader
{
 char read();
}

interface Writer
{
 void write(char);
}

void Copy(Reader r, Writer w)
{
  int c;
  while((c = r.read()) != EOF)
    w.write(c);
}
```

Las dependencias han sido invertidas:
Copy() depende de los interfaces (abstracciones) y los lectores y escritores específicos también.

3. Principio de inversión de dependencias

- **Principio de inversión de dependencias**

A. Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de abstracciones.

B. Las abstracciones no deben depender de detalles específicos. Éstos deben depender de las abstracciones.

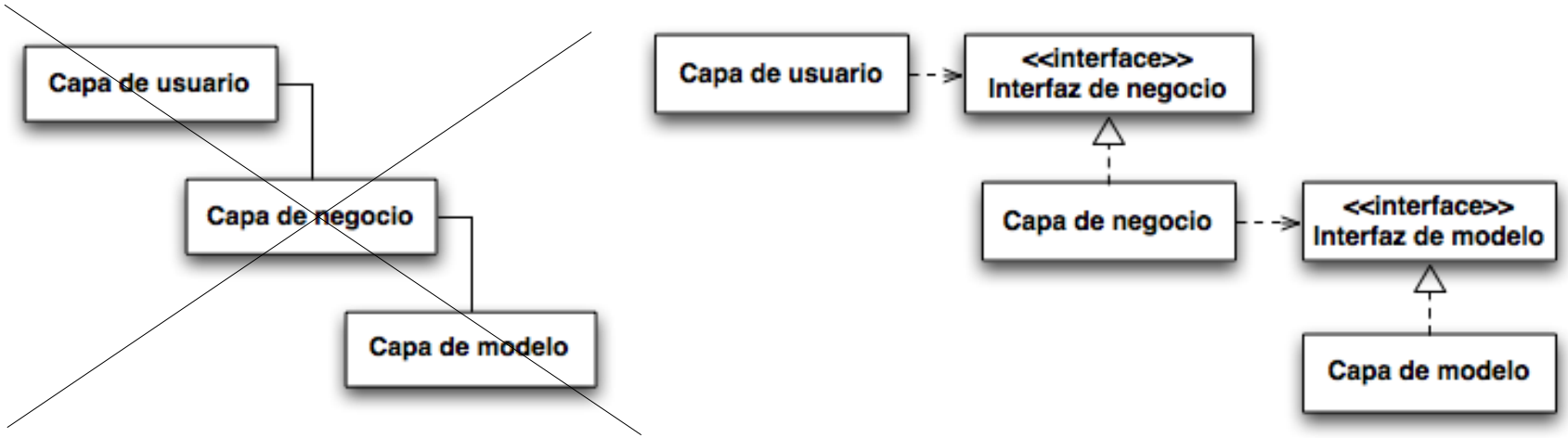
En este principio está basado el diseño de 'frameworks'.

3. Principio de inversión de dependencias

- **Principio de inversión de dependencias**

Según Booch(*):

"Las arquitecturas orientadas a objetos bien estructuradas están diseñadas por capas, donde cada capa porporciona un conjunto coherente de servicios a través de una interfaz bien definida."

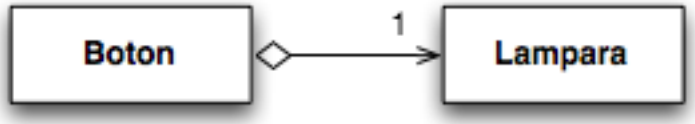


(*) Object Solutions, Grady Booch, Addison Wesley, 1996, p54

3. Principio de inversión de dependencias

- **Ejemplo: el Botón y la Lámpara**

La inversión de dependencias se puede aplicar allí donde una clase envía un mensaje a otra.




```
class Lampara {
  public void encender() {}
  public void apagar() {}
}
```

(No es posible reusar Boton para controlar un Motor, p. ej.)

```
class Boton {
  public Boton(Lampara l)
    { lamp = l;}
  public void detectar() {
    boolean botonOn = getEstado();
    if (botonOn)
      lamp.encender();
    else
      lamp.apagar();
  }
  private Lampara lamp;
}
```

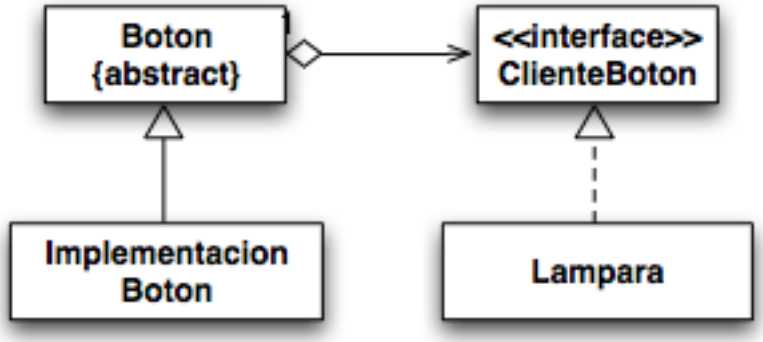
3. Principio de inversión de dependencias

- **Ejemplo: el Boton y la Lampara**

Para aplicar el principio de inversión de dependencias, es necesario encontrar la abstracción subyacente en el diseño.

En el ejemplo Boton/Lampara la abstracción consiste en detectar un estado On/Off y pasar esta información a otro objeto.

El mecanismo usado para detectar el estado, o el tipo de objeto a controlar (la lámpara) es irrelevante.



Ejercicio: Escribe código en Java para este diseño que realice la misma función que el de la página anterior.

3. Principio de inversión de dependencias

- **Conclusiones**

El principio de inversión de dependencias implica la separación de interfaz e implementación en distintos módulos.

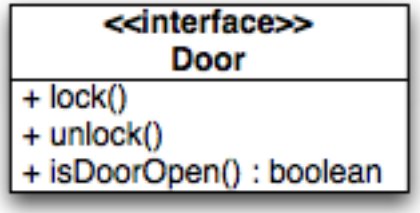
Corolario: Las clases abstractas no deben depender de clases concretas. Las clases concretas son las que deben tener dependencias de clases abstractas

4. Principio de segregación de interfaz

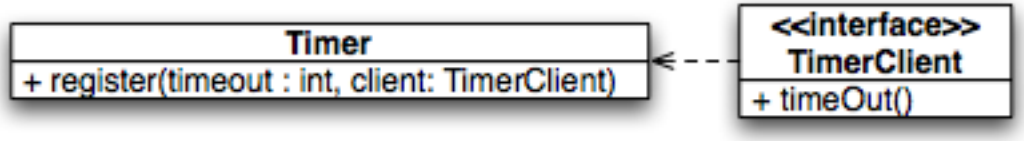
- **Interfaces 'gruesas'**
  - Son aquellas que tienen muchos métodos públicos. Suelen indicar una falta de cohesión en el interfaz.
  - Se pueden dividir en grupos de métodos, cada uno de los cuales sirve a un grupo diferente de clientes (diferentes responsabilidades).
  - El principio de segregación de interfaz admite que hay objetos que requieren interfaces no cohesivas, pero sugiere que este interfaz no tiene que estar definido en una única clase.
  - El código cliente manipula estos objetos a través de diferentes clases abstractas o interfaces bien cohesionados.

4. Principio de segregación de interfaz

- **Polución de interfaz**
  - Imaginemos un sistema de seguridad. En éste hay puertas (Door) que pueden estar abiertas o cerradas, y que saben si están abiertas o cerradas:

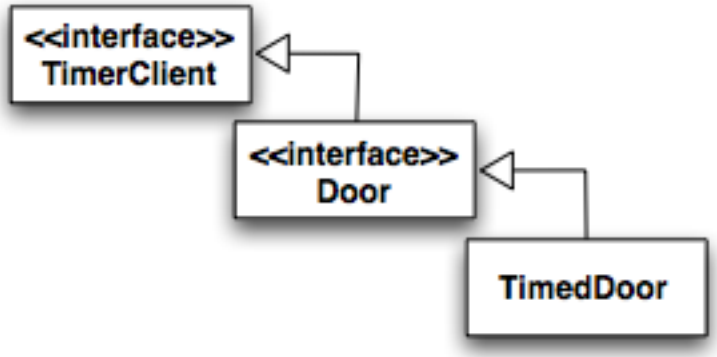


  - Supongamos que hay un tipo de puertas (TimedDoor) que hacen sonar una alarma cuando la puerta permanece abierta demasiado tiempo. Para hacerlo, la puerta se comunica con otro objeto (Timer), el cual invoca a un método timeOut() cuando el tiempo se agota. Para ello, la puerta debe implementar el interface TimerClient y registrarse en el objeto Timer.



4. Principio de segregación de interfaz

- **Polución de interfaz**
  - Solución común:



  - Problemas: no todos los tipos de puertas necesitan un Timer. ëstas se verán obligadas a proporcionar una implementación vacía del método timeOut().
  - Esto se conoce como el síndrome de la polución de interfaz: la interfaz de Door ha sido contaminada por un interfaz que sólo necesitan algunas de sus subclases, conviertiéndose en un interfaz 'grueso'.
  - Todos los clientes de Door se ven afectados por el cambio de interfaz.

4. Principio de segregación de interfaz

- **El principio de segregación de interfaz**

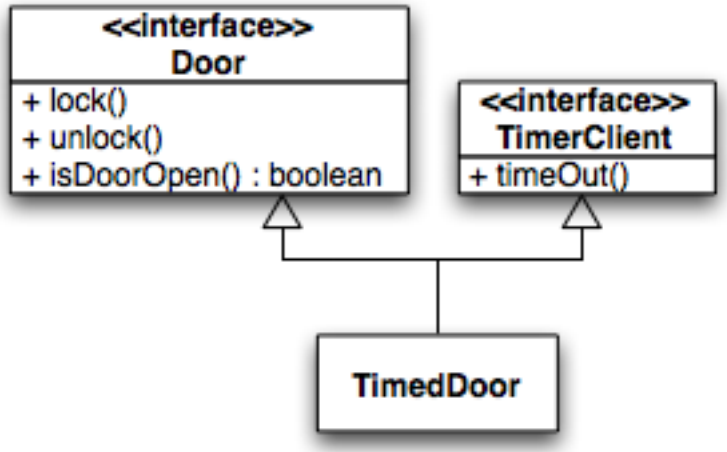
> El código cliente no debe ser forzado a depender de interfaces que no utiliza.

¿cómo segregar interfaces cuando deben estar juntos? (como en el caso de Door)

- Los clientes de un objeto no necesariamente acceden a él a través de la interfaz de la clase del objeto.
  - P. ej.: Separación mediante herencia de interfaz múltiple

4. Principio de segregación de interfaz

- **Separación mediante herencia de interfaz múltiple**



- Los clientes tanto de Door como de TimerClient pueden hacer uso de TimedDoor, pero no tienen ninguna dependencia de ella. Usan el mismo objeto a través de interfaces diferentes.

4. Principio de segregación de interfaz

- **Conclusiones**
  - Los interfaces 'gruesos' o 'hinchados' conducen a acoplamientos entre clientes que de otra forma deberían estar aislados (desacoplados) entre sí.
  - Estos interfaces pueden segregarse en diferentes clases abstractas o interfaces que rompen el acoplamiento no deseado entre los clientes.

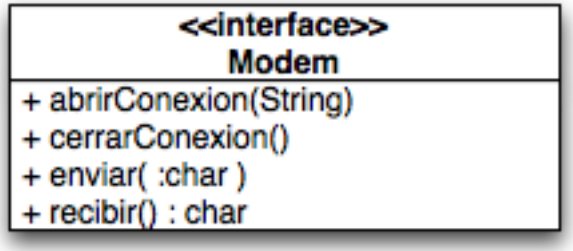
5. Principio de responsabilidad única

Nunca debe existir más de una razón para que una clase cambie.

- Cada responsabilidad asociada a una clase es una razón para que la clase cambie.
- Si una clase asume más de una responsabilidad, entonces existirá más de una razón para que la clase cambie.
- Si una clase asume dos o más responsabilidades, entonces estas se acoplan: cambios en una de ellas pueden impedir satisfacer alguna de las otras → Fragilidad

5. Principio de responsabilidad única

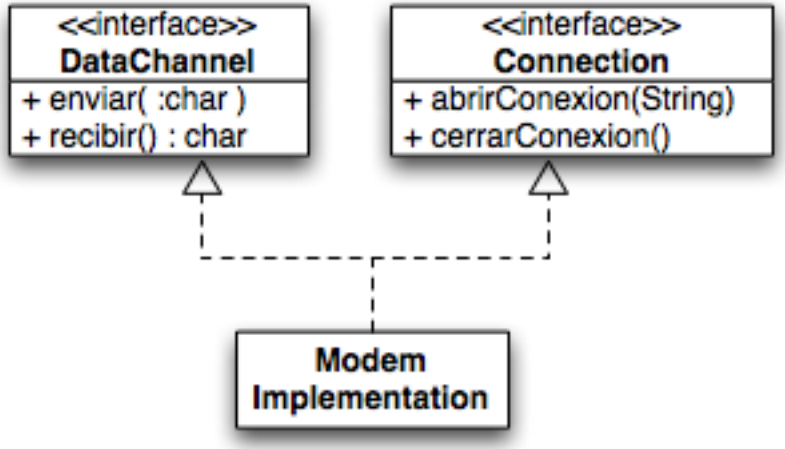
- Ejemplo: Interface Modem



- Parece un diseño razonable, pero Modem está asumiendo dos responsabilidades diferentes.
  - 1. Gestión de la conexión (abrirConexion/cerrarConexion)
  - 2. Comunicación de datos (enviar/recibir)
- Ambos grupos de métodos pueden cambiar por razones diferentes
- Además, son invocados por partes diferentes de la aplicación, que también cambiarán por razones diferentes.

5. Principio de responsabilidad única

- Ejemplo: Interface Modem
  - Este diseño separa las responsabilidades en interfaces diferentes:



  - Las dos responsabilidades son re-acopladas en *ModemImplementation*. Sin embargo, separando los interfaces hemos separado los conceptos, de forma que ningún código cliente (con la posible excepción de algún método factoría) depende de dicha clase, sino de los interfaces desacoplados.

5. Principio de responsabilidad única

- Conclusiones
  - El principio de responsabilidad única es uno de los más simples, y a la vez de los más difíciles de cumplir correctamente
  - Unir responsabilidades es algo que hacemos de forma natural.
  - Encontrarlas y separarlas es, en su mayor parte, de lo que realmente trata el diseño de software.

Principios de diseño orientado a objetos

- Los principios vistos hasta ahora tienen que ver con el diseño de clases.
- Por sus iniciales en inglés se les conoce como el conjunto de principios **SOLID**:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

- Existen otros principios de diseño OO de más alto nivel que tratan, por ejemplo, de la cohesión y acoplamiento a nivel de paquetes.

Un vistazo a los patrones de diseño

- Cuando nos enfrentamos a un nuevo problema ¿dónde buscamos inspiración?
- La mayoría busca soluciones a problemas previos que tienen características similares
- **Patrones de diseño**
  - Un patrón de diseño es una solución a un problema de diseño.
  - Para que una solución sea considerada un patrón debe poseer ciertas características.
    - Se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
    - Debe ser reutilizable, lo que significa que es aplicable a problemas de diseño similares en distintas circunstancias.

Un vistazo a los patrones de diseño

- Los patrones de diseño aceleran el proceso de encontrar una solución, eliminando la necesidad de reinventar soluciones ya probadas y bien conocidas.

- Muchos patrones tienen que ver con el concepto de dependencia, e intentan aumentar la cohesión de la solución al tiempo que reducen su acoplamiento.

Un vistazo a los patrones de diseño

- Un ejemplo simple: el patrón **Adaptador**
  - Un adaptador es un componente que se usa para conectar un cliente (que necesita algún servicio) con un servidor (que proporciona este servicio)
  - El problema es que el cliente requiere un determinado interfaz. Aunque el servidor proporciona la funcionalidad deseada, no soporta el interfaz requerido.
  - El adaptador cambia la interfaz del servidor, delegando el trabajo en éste.

Un vistazo a los patrones de diseño

- Un ejemplo simple: el patrón **Adaptador**

```
class MiAdaptadorColeccion implements Collection {

  public boolean isEmpty ()
    { return data.cuantos() == 0; }
  public int size ()
    { return data.cuantos(); }
  public void add (Object nuevo)
    { data.colocarAlFinal(nuevo); }
  public boolean contains (Object test)
    { return data.busca(test) != null; }
  // … etc

  private Vectorcito data = new Vectorcito();
}
```

- 'Vectorcito' es una clase contenedora que no implementa la interfaz de Collection
- Los adaptadores se usan a menudo para conectar software de diferentes fabricantes.

Bibliografía

- Robert C. Martin. ***Principles of object oriented design***
  http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

- T.C. Lethbridge, R. Laganiere. ***Object-oriented software engineering : practical software development using UML and Java***
  - *Sec. 9.1 (el proceso de diseño)*

- T. Budd. ***An Introduction to Object-oriented Programming, 3rd ed.***
  - *Cap 23 (acoplamiento y cohesion)*
  - *Cap 24 (patrones de diseño)*

- E. Gamma et al. ***Design Patterns. Elements of Reusable Object-Oriented Software.***