Diseño de clases

¿Cómo sabemos si nuestro diseño es correcto?

Existen algunos síntomas que nos indican
que el diseño de un sistema es bastante pobre:

- RIGIDEZ (las clases son difíciles de cambiar)
- FRAGILIDAD (es fácil que las clases dejen de funcionar)
- INMOVILIDAD (las clases son difíciles de reutilizar)
- VISCOSIDAD (resulta difícil usar las clases correctamente)
- COMPLEJIDAD INNECESARIA (sistema "sobrediseñado")
- REPETICIÓN INNECESARIA (abuso de "copiar y pegar")
- OPACIDAD (aparente desorganización)

Afortunadamente, también existen algunos principios heurísticos
que nos ayudan a eliminar los síntomas arriba enumerados:

- **Principio de responsabilidad única**
- **Principio abierto-cerrado**
- **Principio de sustitución de Liskov**
- **Principio de inversión de dependencias**
- **Principio de segregación de interfaces**

Descripción de los síntomas de un diseño "mejorable"

Rigidez

El sistema es difícil de cambiar porque cualquier cambio,
por simple que sea, fuerza otros muchos cambios en cascada.

> *¿Por qué es un problema?*
> *"Es más complicado de lo que pensé"*
> Cuando nos dicen que realicemos un cambio, puede que nos encontremos con que hay que hacer más cosas de las que, en principio, habíamos pensado que harían falta.

Cuantos más módulos haya que tocar para hacer un cambio, más rígido es el sistema.

Fragilidad

Los cambios hacen que el sistema deje de funcionar
(incluso en lugares que, aparentemente,
no tienen nada que ver con lo que hemos tocado).

> *¿Por qué es un problema?*
> Porque la solución de un problema genera nuevos problemas…

Conforme la fragilidad de un sistema aumenta, la probabilidad de que un cambio ocasione nuevos quebraderos de cabeza también aumenta.

Inmovilidad

La reutilización de componentes requiere demasiado esfuerzo.

> *¿Por qué es un problema?*
> Porque la reutilización es siempre la solución más rápida.

Aunque un diseño incorpore elementos potencialmente útiles, su inmovilidad hace que sea demasiado difícil extraerlos.

Viscosidad

Es más difícil utilizar correctamente lo que ya hay implementado que hacerlo de forma incorrecta (e, incluso, que reimplementarlo).

> *¿Por qué es un problema?*
> Cuando hay varias formas de hacer algo, no siempre se elige la mejor forma de hacerlo y el diseño tiende a degenerarse.

La alternativa más evidente y fácil de realizar ha de ser aquélla que preserve el estilo del diseño.

Complejidad innecesaria

El diseño contiene infraestructura que no proporciona beneficios.

> *¿Por qué es un problema?*
> A veces se anticipan cambios que luego no se producen, lo que conduce a *más* código (que hay que depurar y mantener)

La complejidad innecesaria hace que el software sea más difícil de entender.

Repetición innecesaria

"Copiar y pegar" resulta perjudicial a la larga.

> *¿Por qué es un problema?*
> Porque el mantenimiento puede convertirse en una pesadilla.

El código duplicado debería unificarse bajo una única abstracción.

Opacidad

Código enrevesado difícil de entender.

> *¿Por qué es un problema?* Porque la "entropía" del código tiende a aumentar si no se toman las medidas oportunas.

Escribimos código para que otros puedan leerlo (y entenderlo).

El principio de responsabilidad única

Tom DeMarco, 1979: *"Structured Analysis and System Specification"*

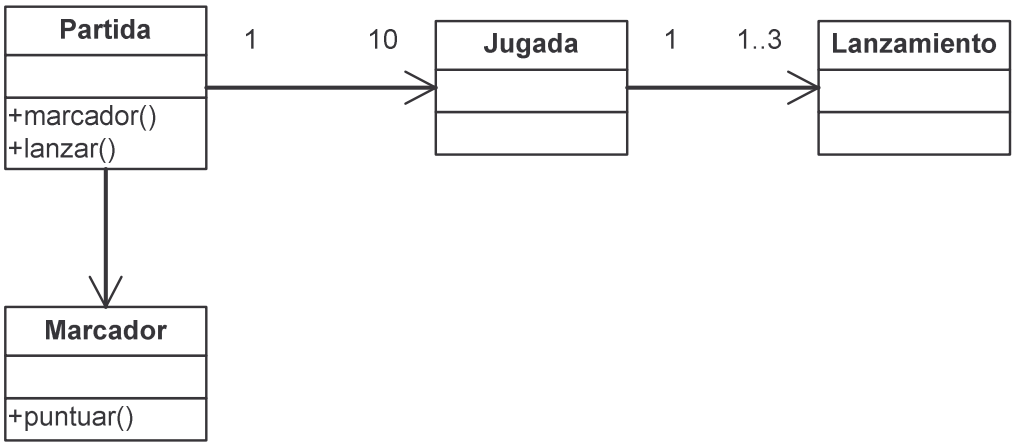
> **Una clase debe tener un único motivo para cambiar.**

Por ejemplo, podemos crear una clase `Partida` para representar una partida de bolos (que consta de 10 jugadas):

La clase `Partida` tiene aquí dos responsabilidades diferentes:

- Mantener el estado actual de la partida (esto es, cuántos lanzamientos llevamos realizados y cuántos nos quedan)
- Calcular nuestra puntuación (siguiendo las reglas oficiales del juego: 10 puntos por pleno [*strike*], etc.).

En una situación así, puede interesarnos separar el cálculo de la puntuación del mantenimiento del estado de la partida:



De esta forma, podríamos utilizar nuestra clase `Partida` para distintas variantes del juego (si es que existiesen) o incluso para otros juegos (¿la petanca?).

En este contexto,
una responsabilidad equivale a “una razón para cambiar”.

Si se puede pensar en más de un motivo por el que cambiar una clase, entonces la clase tiene más de una responsabilidad.

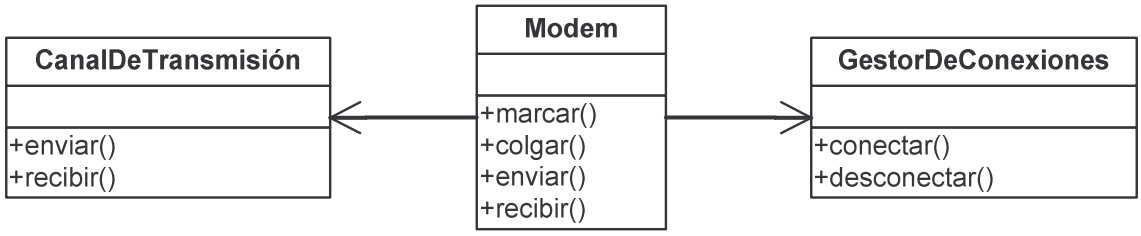
Ejemplo

```
public class Modem
{
  public void marcar (String número) …
  public void colgar () …

  public void enviar (String datos) …
  public String recibir () …
}
```

La clase Modem tiene aquí dos responsabilidades:

- Encargarse de la gestión de las conexiones.
- Enviar y recibir datos.



`Modem` sigue teniendo dos responsabilidades y sigue siendo necesaria en nuestra aplicación, pero ahora nada depende de `Modem`:
Sólo el programa principal ha de conocer la existencia de `Modem`.

Si la forma de hacerse cargo de una de las responsabilidades sabemos que nunca varía, no hay necesidad de separarla
(se trataría de complejidad innecesaria en nuestro diseño):

Una razón de cambio sólo es una razón de cambio
si el cambio llega a producirse realmente.

El principio abierto-cerrado

Bertrand Meyer, 1997: *"Construcción de software orientado a objetos"*

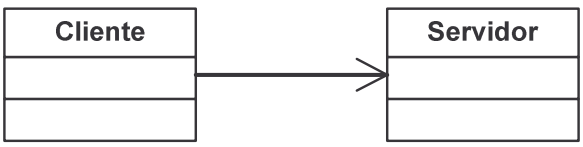
Los módulos deben ser a la vez abiertos y cerrados:
abiertos para ser extendidos y cerrados para ser usados.

1. Debe ser posible extender un módulo para ampliar su conjunto de operaciones y añadir nuevos atributos a sus estructuras de datos.

2. Un módulo ha de tener un interfaz estable y bien definido (que no se modificará para no afectar a otros módulos que dependen de él).

Si se aplica correctamente este principio,
los cambios en un sistema se realizan siempre añadiendo código, sin tener que modificar la parte del código que ya funciona.

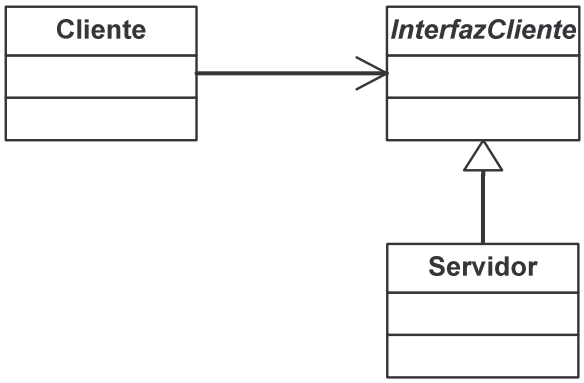
El sencillo diseño siguiente no se ajusta al principio abierto-cerrado:



- El `Cliente` usa los servicios de un `Servidor` concreto, al cual está ligado. Si queremos cambiar el tipo de servidor, hemos de cambiar el código del cliente.

Aunque pueda resultar paradójico, se puede cambiar lo que hace un módulo sin cambiar el módulo: la clave es la **abstracción**.

El siguiente diseño sí se ajusta al principio abierto-cerrado:



- `InterfazCliente` es una clase base utilizada por el cliente para acceder a un servidor concreto: Los clientes acceden a servidores concretos que implementen el interfaz definida por la clase `InterfazCliente`.

- Si queremos que un cliente utilice un servidor diferente, basta con crear una clase nueva que también implemente la interfaz de `InterfazCliente` (esto es, la clase `Cliente` no habrá que modificarla).

- El trabajo que tenga que realizar el `Cliente` puede describirse en función del interfaz genérico expuesto por `InterfazCliente`, sin recurrir a detalles de servidores concretos.

¿Por qué `InterfazCliente` y no `ServidorGenérico`?

Porque las clases abstractas (más sobre ellas más adelante) están más íntimamente relacionadas a sus clientes que a las clases concretas que las implementan.

El principio abierto-cerrado
es clave en el diseño orientado a objetos.

Si nos ajustamos a él, obtenemos los mayores beneficios
que se suelen atribuir a la orientación a objetos
(flexibilidad, mantenibilidad, reutilización).

No obstante,
pese a que la abstracción y el polimorfismo lo hacen posible,

el principio abierto-cerrado
no se garantiza simplemente con utilizar
un lenguaje de programación orientado a objetos.

De la misma forma,
tampoco tenemos que crear abstracciones arbitrariamente:

- No todo tiene que ser abstracto:
  El código debe acabar haciendo algo concreto.

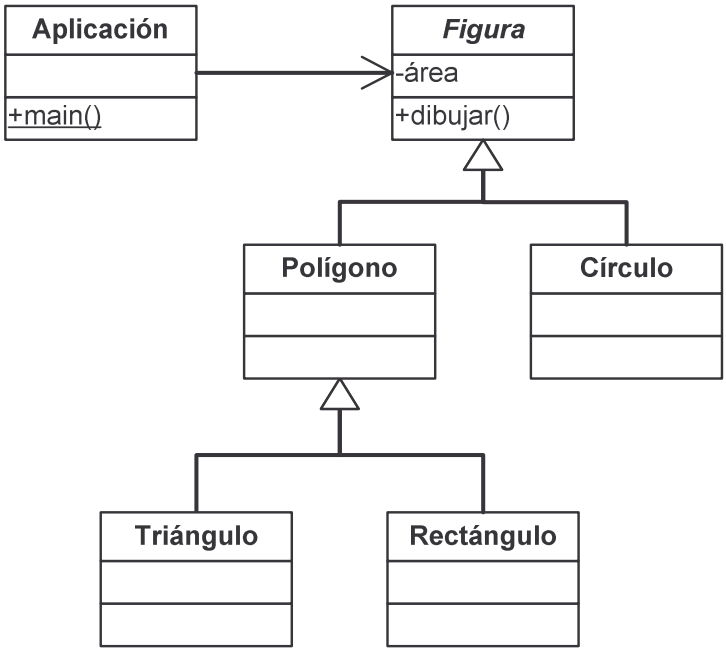
- Las abstracciones se han de utilizar
  en las zonas que exhiban cierta propensión a sufrir cambios.

El principio de sustitución de Liskov

Barbara Liskov: *"Data abstraction and hierarchy"*. SIGPLAN Notices, 1988

Los tipos base siempre se pueden sustituir por sus subtipos.

Si S es un subtipo de T,
cualquier programa definido en función de T
debe comportarse de la misma forma con objetos de tipo S.



La importancia de este principio
se hace evidente cuando deja de cumplirse:

```
public class Figura
{
  …
  public void dibujar ()
  {
    if (this instanceof Polígono)
       dibujarPolígono();
    else if (this instanceof Círculo)
       dibujarCírculo();
  }
  …
}
```

El método `dibujar` viola el principio de sustitución de Liskov, porque ha de ser consciente de cualquier subtipo de `Figura` que creemos en nuestro sistema (p.ej. ¿qué sucede si también tenemos que trabajar con elipses?).

De hecho, también se viola el *principio abierto-cerrado*.

La solución: Utilizar polimorfismo
Cada tipo de figura será responsable de implementar el método `dibujar`.

```
public class Polígono extends Figura
  …
  public void dibujar ()
  …

public class Círculo extends Figura
  …
  public void dibujar ()
  …
```

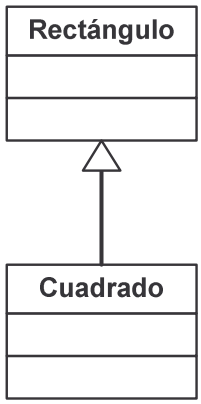
Veamos un caso algo más sutil:
Inicialmente trabajamos con rectángulos…

```
public class Rentángulo
{
  private double anchura;
  private double altura;

  public double getAnchura () …
  public double getAltura () …

  public void setAnchura () …
  public void setAltura () …
}
```

Por lo que sea, también hemos de manipular cuadrados…



La implementación empieza a crear “dificultades”…

```
public class Cuadrado extends Rentángulo
{
  public void setAnchura (double x)
  {
    super.setAnchura(x);
    super.setAltura(x);
  }

  public void setAltura (double x)
  {
    super.setAnchura(x);
    super.setAltura(x);
  }
}
```

Ahora, consideremos el siguiente método:

```
…
void f (Rectángulo rect)
{
  rect.setAnchura(32);
}
…
```

¿Qué sucede si llamamos a este método con un cuadrado?

> La llamada a `setAnchura`
> establece también la altura del “rectángulo”
> (algo aparentemente correcto).

Pero, ¿y si el método hace algo diferente?

```
…
void g (Rectángulo rect)
{
  rect.setAnchura(4);
  rect.setAltura(5);
  Assert.assertEquals( rect.getArea(), 20);
}
…
```

Obviamente, el código anterior fallará
si el método recibe un cuadrado en vez de un triángulo `L`

¿Por qué falla la implementación?

Aunque el modelo del cuadrado como un tipo particular de rectángulo parezca razonable, desde el punto de vista del programador del método `g`, ¡un cuadrado no es un rectángulo!

Las relaciones de herencia se han de utilizar para heredar comportamiento (el comportamiento que los clientes de una clase pueden asumir y en el que confían).

Formalmente,
la interfaz de una clase define su contrato.

El contrato se especifica declarando
las precondiciones y las postcondiciones asociadas a cada método
(esto es, lo que verificamos con las pruebas de unidad con JUNIT):

- Las **precondiciones** son las condiciones que
  se han de cumplir antes de la llamada al método.

- Las **postcondiciones** son las condiciones que
  se verifican tras la ejecución del método.

En el caso de la clase `Rectángulo`,
las postcondición implícita de `setAnchura` es

```
(anchura == x) && (altura == old.altura)
```

donde `old` hace referencia al estado del rectángulo antes de la llamada.

Para la clase `Cuadrado`, no obstante,
la postcondición asociada a `setAnchura` es

```
(anchura == x) && (altura == x)
```

Nuestra implementación de `setAnchura` para la clase `Cuadrado`
es incorrecta porque impone un postcondición distinta
a la impuesta por `setAnchura` en su clase base `Rectángulo`.

> La redefinición de un método en una clase derivada sólo puede
> reemplazar la precondición original por una más débil
> y la postcondición original por otra más fuerte (restrictiva).

¿Por qué? Porque cuando se usa una clase base, sólo se conocen las precondiciones y postcondiciones asociadas a la clase base.

El principio de inversión de dependencias

Robert C. Martin: *C++Report*, 1996

a) Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

b) Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

La programación estructurada
tiende a crear estructuras jerárquicas en las que

el comportamiento de los módulos de alto nivel
(p.ej. el programa principal)
depende de detalles de módulos de un nivel inferior
(p.ej. la opción del menú seleccionada por el usuario).

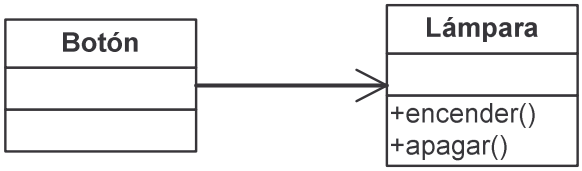
La estructura de dependencias en una aplicación orientada a objetos bien diseñada suele ser la inversa.

Si se mantuviese la estructura tradicional:

- Cuando se cambiasen los módulos de nivel inferior, habría que modificar los módulos de nivel superior.
- Además, los módulos de nivel superior resultarían difíciles de reutilizar si dependiesen de módulos inferiores.

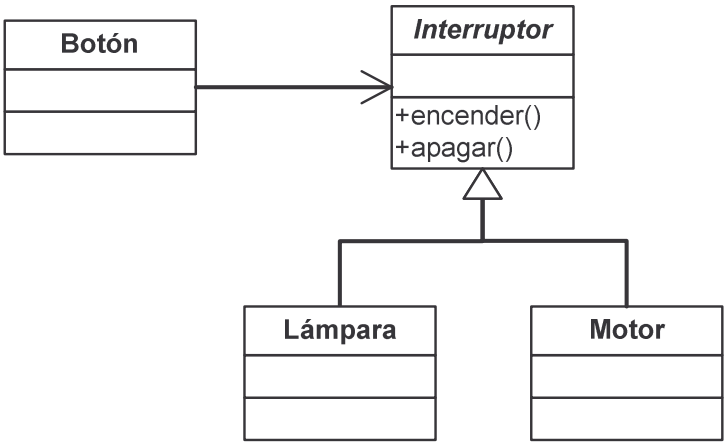
Ejemplo

La inversión de dependencias de puede realizar siempre que una clase envía un mensaje a otra y deseamos eliminar la dependencia



Tal como está diseñado, no será posible utilizar el botón para encender, por ejemplo, un motor (ya que el botón depende directamente de la lámpara que enciende y apaga).

Para solucionar el problema,
volvemos a hacer uso de nuestra capacidad de abstracción:



Ahora podremos utilizar el botón
para cualquier cosa que se pueda encender y apagar.