PRINCIPIOS DE DISEÑO

ORIENTADO A OBJETOS

Principios SOLID

Los principios SOLID de Uncle Bob Martin nos dicen:

- Cómo organizar en **módulos** (en OO, **clases**) las estructuras de datos y las funciones
- Cómo deben quedar *interconectadas* las clases (vía **dependencias**)

El objetivo de SOLID es crear estructuras software de nivel intermedio que sean:

- *flexibles*: tolerantes a los cambios
- *poco complejas*: fáciles de comprender
- *reutilizables*: la base de componentes útiles para muchos sistemas software

En C++: Breaking Dependencies: The SOLID Principles by Klaus Iglberger

SRP: *Single Responsibility Principle*

Principio de responsabilidad única

> A class should have only one reason to change
> — Bob Martin

- Una clase que modela múltiples aspectos genera acoplamiento entre los distintos aspectos
- Un cambio en algún aspecto obligará a cambios accidentales en los clientes que no dependen de dicho aspecto

SRP es lo mismo que el principio de **cohesión** de DeMarco

SRP es aplicación directa de la ley de Conway:

> Any organization that designs a system (...) will produce a design whose structure is a copy of the organization's communication structure.
> — M. Conway, *Datamation*, April 1968

- Cuando se diseña software, hace falta conocer los grupos/equipos/roles a los que éste sirve, y dividir el sistema en componentes separados, de forma similar a como estos grupos de personas se comunican normalmente en la vida real.
- Es necesario tener conocimiento del **dominio** para poder dividir bien las responsabilidades
- Tiene que ver con la **variabilidad** de los requisitos

CASO PRÁCTICO 4

Figuras geométricas

Ejemplo: Shapes versión 1 en Java

```
package shapes;

interface Shape {
  double area();
  void draw();
}

class Point {
  double getX() {...}
  double getY() {...}
}

abstract class Polygon implements Shape {
  Point getVertex(index i) {...}
  void draw() {...}
  String toString() {...}
}

class Triangle extends Polygon {
  double area() {...}
}
abstract class RectParallelogram extends Polygon {
  double area() {...}
}
```

```
class Square extends RectParallelogram {...}

class Rectangle extends RectParallelogram {...}

abstract class ClosedCurve implements Shape {...}

class Circle extends ClosedCurve {
  double getRadius() {...}
  Point getCenter() {...}
  double area() {...}
  void draw() {...}
  String toString() {...}
}

class Ellipse extends ClosedCurve {
  double getApogeeRadius() {...}
  double getPerigeeRadius() {...}
  Point getFocus1() {...}
  Point getFocus2() {...}
  Point getCenter() {...}
  double area() {...}
  void draw() {...}
  String toString() {...}
}
```

Preguntas

- ¿Cuántas responsabilidades tienen las clases que implementan la interfaz `Shape`?
- ¿Cuáles son estas responsabilidades?
- ¿Qué parte no cumple SRP en el ejemplo?

Respuestas

- Dos responsabilidades: geometría computacional + dibujo en pantalla
- Todas las figuras tienen métodos `draw` y `toString` (dibujar en pantalla) además del método `area` que calcula el área (geometría computacional) $\rightarrow$ Violación del SRP

Solución

Patrón de diseño **Visitor**

Ejercicio

- Buscar información de los patrones *ActiveRecord* y *Data Access Object (DAO)*.
- Discutir si cumplen o violan el SRP.

Ejemplo en C++: Circle (version original)

```
class Circle
{
  public:
    explicit Circle (double rad ) : radius { rad }, //... remaining data members
    {}

    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate (Vector3D const& );
    void rotate ( Quaternion const& );

    void draw ( Screen& s, /*...*/ );
    void draw ( Printer& p, /*...*/ );
    void serialize ( ByteStream& bs, /*...*/ );
    //...

  private:
    double radius;
    ///... remaining data members
}
```

Ejemplo en C++: Circle (refactor SRP 1)

```
class Circle
{
  public:
    explicit Circle (double rad ) :
         radius { rad },
         //... remaining data members
    {}

    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate (Vector3D const& );
    void rotate ( Quaternion const& );

  private:
    double radius;
    ///... remaining data members
};
```

```
class CircleRenderer
{
  public:
    virtual ~CircleRenderer() = default;
    virtual void draw ( Circle const&, Screen& s,
                        /*...*/ ) = 0;
    virtual void draw ( Circle const&, Printer& p,
                        /*...*/ ) = 0;
};

class CircleSerializer
{
  public:
    void serialize ( Circle const&, ByteStream& bs,
                     /*...*/ ) const;
};
```

Ejemplo en C++: Circle (refactor SRP 2)

```
class Circle
{
  public:
    explicit Circle (double rad ) : radius { rad }, /* ...*/ {}

    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate (Vector3D const& );
    void rotate ( Quaternion const& );

  private:
    double radius;
    ///... remaining data members
};

// Same namespace as Circle
void draw ( Circle const&, Screen& s, /*...*/ );
void draw ( Circle const&, Printer& p, /*...*/ );
void serialize ( Circle const&, ByteStream& bs, /*...*/ );
//...
```

OCP: *Open-Closed Principle*

Principio de Abierto-Cerrado

> Toda clase, módulo, aspecto o función debe quedar abierto para extensiones pero cerrado para modificaciones
>
> —B. Meyer, Object Oriented Software Construction

Para que un sistema software sea fácil de cambiar, debe diseñarse para que permita cambiar su comportamiento añadiendo código, no cambiando código existente.

- Si un cambio en un sitio origina una cascada de cambios en otros puntos del sistema, el resultado es un sistema frágil y rígido
- Es difícil averiguar todos los puntos que requieren cambios
- OCP mediante delegación en vertical (subclases) u horizontal (composición)

Ejemplo: Shapes versión 2 en C++

¿Qué parte no cumple OCP en el ejemplo?

Versión sin objetos

```
enum ShapeType {circle, square};
struct Shape
{
  ShapeType itsType;
};

struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
void DrawCircle(struct Circle*);

struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
```

```
void DrawSquare(struct Square*);

typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
      case square:
        DrawSquare((struct Square*)s);
        break;
      case circle:
        DrawCircle((struct Circle*)s);
        break;
    }
  }
}
```

Problema:

- `DrawAllShapes` no está cerrado para modificaciones cuando aparecen nuevos tipos de `Shape`

Solución

- **Abstracción** (ocultación de la implementación): clase abstracta y métodos polimórficos.
- **Patrones de diseño**: *template method* y/o *strategy*

Aplicando el OCP...

```
public interface Shape
{
  void Draw();
}

public class Square: Shape
{
  public void Draw()
  {
    //draw a square
  }
}

public class Circle: Shape
{
  public void Draw()
  {
    //draw a circle
  }
}
```

```
public void DrawAllShapes(IList shapes)
{
  foreach(Shape shape in shapes)
    shape.Draw();
}
```

- Si queremos ampliar el comportamiento de `DrawAllShapes`, solo tenemos que añadir una nueva clase derivada de `Shape`
- Si se aplica bien OCP, los cambios de un cierto tipo obligan a añadir nuevo código, no a modificar el existente

Ejercicio: Shapes and Circles

Arreglar para que cumpla OCP

```
enum ShapeType
{
  circle,
  square,
  rectangle
};

class Shape
{
  public:
    explicit Shape ( ShapeType t )
      : type { t }
    {}
    virtual ~Shape() = default;
    ShapeType getType() const noexcept;

  private:
    ShapeType type;
};
```

```
class Circle: public Shape
{
  public:
    explicit Circle ( double rad )
      : Shape{ circle }
      , radius { rad }
      , //... remaining data members
    {}

    virtual ~Circle() = default;
    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

  private:
    double radius;
    ///... remaining data members
};

void translate ( Circle&, Vector3D const& );
void rotate ( Circle&, Quaternion const& ) ;
void draw ( Circle const& );
```

```
class Square: public Shape
{
  public:
    explicit Square ( double s )
      : Shape{ square }
      , side { s }
      , // ... remaining data members
    {}

    virtual ~Square() = default;
    double getSide() const noexcept;
    //... getCenter(), getRotation(), ...

  private:
    double side;
    // ... remaining data members
};

void translate ( Square&, Vector3D const& );
void rotate ( Square&, Quaternion const& ) ;
void draw ( Square const& );
```

```
void draw ( std::vector<std::unique_ptr<<Shape>>>) const & shapes )
{
  for ( auto const& s : shapes )
  {
    switch ( s-> getType() )
    {
      case circle:
        draw ( *static_cast<Circle const*>( s.get() ) );
        break;
      case square:
        draw ( *static_cast<Square const*>( s.get() ) );
        break;
      case rectangle:
        draw ( *static_cast<Rectangle const*>( s.get() ) );
        break;
    }
  }
}
```

```
int main()
{
  using Shapes = std::vector<std::unique_ptr<Shape>>;
  // Creating some shapes
  Shapes shapes;
  shapes.push_back (std::make:unique<Circle>( 2.0 ));
  shapes.push_back (std::make:unique<Square>( 1.5 ));
  shapes.push_back (std::make:unique<Circle>( 4.2 ));
  // Drawing all shapes
  draw ( shapes );
}
```

Cierre estratégico

> In general, no matter how *closed* a module is, there will always be some kind of change against which it is not closed. There is no model that is natural to all contexts!
>
> Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close the design, must guess at the kinds of changes that are most likely, and then construct abstractions to protect against those changes.
>
> — Bob C. Martin

Implicaciones arquitectónicas

OCP es un principio más arquitectónico que de diseño de clases y módulos.

Solución al ejercicio: Shapes and Circles

Versión en C++ que cumple el OCP y SRP

```
class Circle : public Shape
{
  public:
    explicit Circle (double rad )
      : radius { rad }
      , //... remaining data members
    {}

    virtual ~Circle() = default;

    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate ( Vector3D const& ) override;
    void rotate ( Quaternion const& ) override;
    void draw () const override;

  private:
    double radius;
    ///... remaining data members
}
```

```
class Square : public Shape
{
  public:
    explicit Square (double s )
      : side { s }
      , //... remaining data members
    {}

    virtual ~Square() = default;

    double getSide() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate ( Vector3D const& ) override;
    void rotate ( Quaternion const& ) override;
    void draw () const override;

  private:
    double side;
    ///... remaining data members
}
```

```
class Shape
{
  public:
    Shape() = default;
    virtual ~Shape() = default;

    virtual void translate ( Vector3D const& ) = 0;
    virtual void rotate ( Quaternion const& ) = 0;
    virtual void draw() const = 0; // check!
};
```

```
void draw ( std::vector<std::unique_ptr<<Shape>>>) const & shapes )
{
  for ( auto const& s : shapes )
  {
      s->draw();
  }
}
```

ISP: *Interface Segregation Principle*

Principo de segregación de interfaces

> Los clientes no deben depender de métodos que no usan.
>
> Bob C. Martin

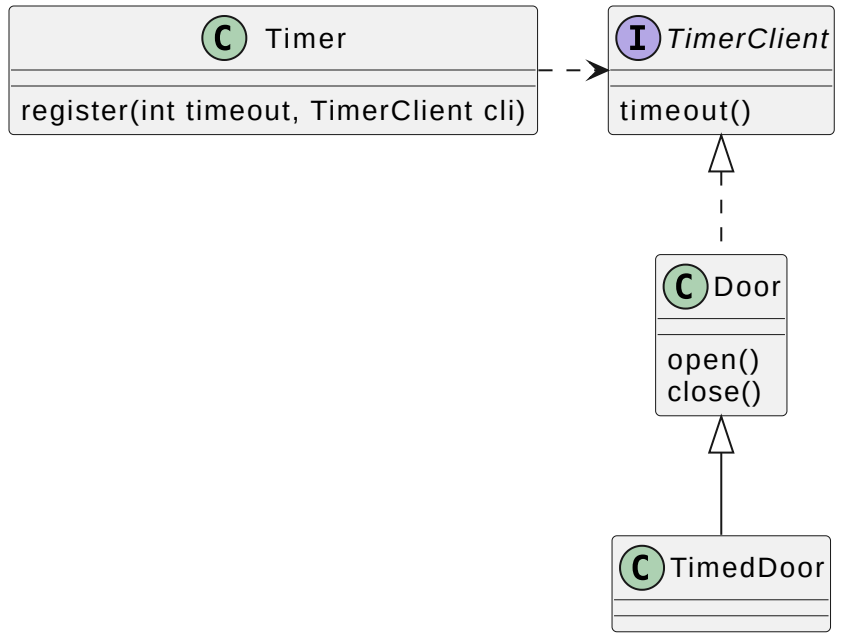
- Las interfaces son para los **clientes**, no para hacer jerarquías
- Evitar interfaces **gruesas** con muchos métodos (descohesionadas)
- Los cambios en los métodos ignorados pueden provocar cambios en un cliente que no los usa

- La interfaz de una clase puede dividirse en **bloques** de métodos relacionados. Unos clientes usan un bloque y otros clientes usan otro bloque. Si un cliente necesita conocer una interfaz no cohesionada, debe hacerlo combinando una o más clases (o mejor, sus interfaces)
- ISP es a las interfaces lo que SRP es a clases y métodos
- Violar el ISP es muy común en lenguajes de tipos estáticos (C++, Java, C#). Los lenguajes dinámicos (Ruby, Scala) ayudan algo más a no violar el ISP (v.g. con los *mixins*)

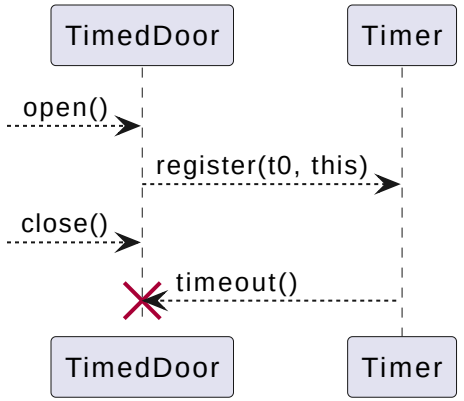
Ejemplo: puertas de seguridad

Una implementación de puertas de seguridad con temporizador `TimedDoor` que hace sonar una alarma cuando la puerta está abierta durante un cierto tiempo.

Diseño:



- `TimedDoor` se comunica con `Timer` para registrar un temporizador
- Cuando salta el temporizador, avisa a un `TimerClient`
- Con la solución diseñada, un `TimerClient` puede registrarse a sí mismo en un `Timer` y recibir de éste un mensaje mediante `timeout()`.



Implementación inicial

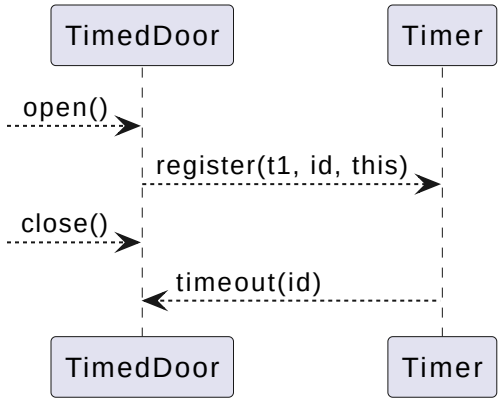
```
public class Timer {
  public void register(int timeout, TimerClient client) {
    /*code*/
  }
}

public interface TimerClient {
    void timeout();
}
```

- Si se cierra la puerta antes de que venza el timeout t_0 y se vuelve a abrir, se registra uno nuevo t_1 antes de que el antiguo haya expirado.
- Cuando el primer temporizador t_0 expira, se produce la llamada a `timeout()` de `TimedDoor` y no debería.
- Así que cambiamos la implementación:

Implementación mejorada

```
public class Timer {
  public void register(int timeout, int timeOutId, TimerClient client) {
    /*code*/
  }
}
public interface TimerClient {
  void timeout(int timeOutID);
}
```

¿En qué ha afectado el **cambio en la implementación** de `TimerClient`?

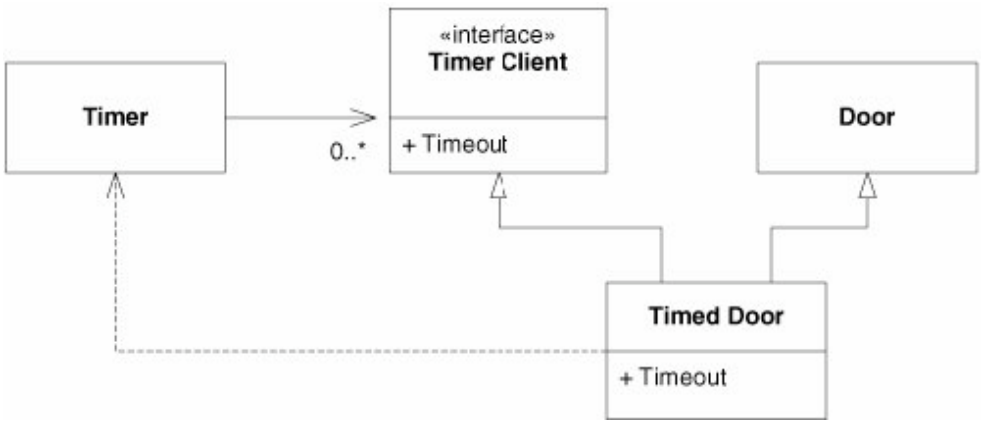


- El cambio afecta a los usuarios de `TimerClient`, pero también a `Door` y a los clientes de `Door` (y no debería)
- El problema es que `Door` depende de `TimerClient` y no todas las variedades de puerta son de seguridad (con temporizador)
- Si hacen falta más variedades de puerta, todas ellas deberán implementar implementaciones degeneradas de `timeout()`
- Las interfaces empiezan a engrosarse. Esto puede acabar violando también el LSP

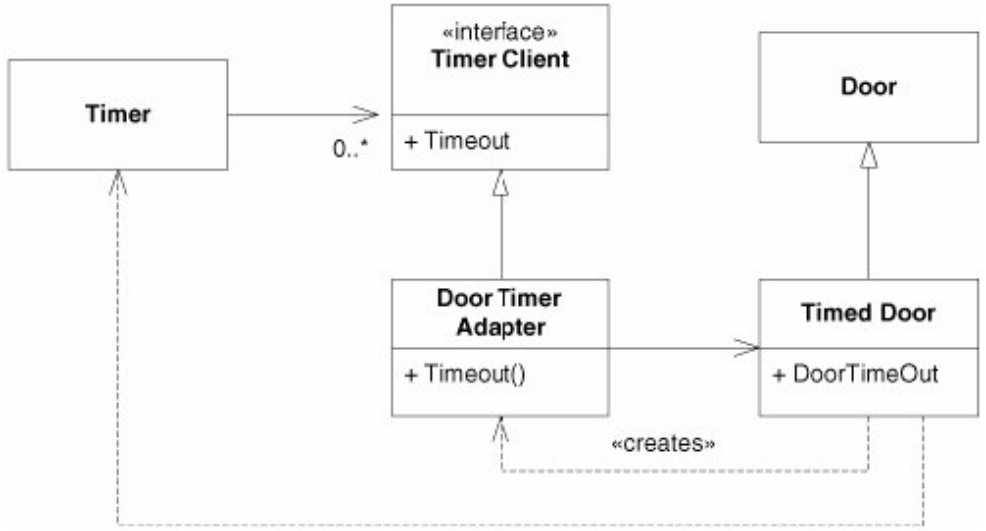
Rediseño: puertas de seguridad

Delegación a través del patrón adapter

- Adaptador de clases (herencia):



- Adaptador de objetos (composición):



Example: Shapes and Circles (1 de 2)

```
class Circle;
class Square;

class DrawStrategy
{
  public:
    virtual ~DrawStrategy() {}

    virtual void draw ( const Circle& circle ) const = 0;
    virtual void draw ( const Square& square ) const = 0;
};

class Shape
{
  public:
    Shape() = default;
    virtual ~Shape() = default;

    virtual void translate ( Vector3D const& ) = 0;
    virtual void rotate ( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};
```

Example: Shapes and Circles (2 de 2)

```
class Circle : public Shape
{
  public:
    explicit Circle ( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius { rad }
      , //... remaining data members
      , drawing { std::move(ds) }
    {}

    virtual ~Circle() = default;

    double getRadius() const noexcept;
    //... getCenter(), getRotation(), ...

    void translate ( Vector3D const& ) override;
    void rotate ( Quaternion const& ) override;
    void draw () const override;

  private:
    double radius;
    ///... remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

class Square: public Shape
{
  //...
}
```

Aplicación de OCP y SRP

Ejemplo: Shapes versión 1 en Java (misma versión que en SRP)

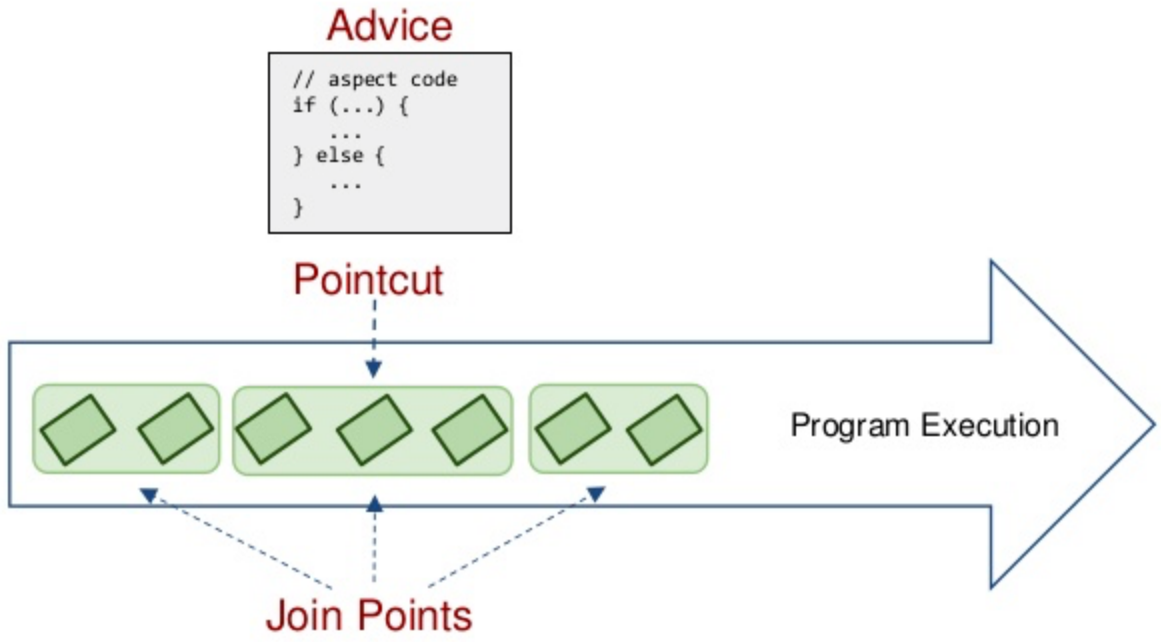
```
interface Shape {
  double area();
  void draw();
}

class Point {
  double getX() {...}
  double getY() {...}
}

abstract class Polygon implements Shape {
  Point getVertex(index i) {...}
  void draw() {...}
  String toString() {...}
}

class Triangle extends Polygon {
  double area() {...}
}

abstract class RectParallelogram extends Polygon {
  double area() {...}
}
```

```
class Square extends RectParallelogram {...}
class Rectangle extends RectParallelogram {...}

abstract class ClosedCurve implements Shape {...}

class Circle extends ClosedCurve {
  double getRadius() {...}
  Point getCenter() {...}
  double area() {...}
  void draw() {...}
  String toString() {...}
}

class Ellipse extends ClosedCurve {
  double getApogeeRadius() {...}
  double getPerigeeRadius() {...}
  Point getFocus1() {...}
  Point getFocus2() {...}
  Point getCenter() {...}
  double area() {...}
  void draw() {...}
  String toString() {...}
}
```

- Las funcionalidades para pintar (`draw`) y para imprimir (`toString`) pueden descohesionar las clases y atentar contra OCP y SRP.
- Saquémoslas fuera utilizando **aspectos**...

Orientación a aspectos

La **orientación a aspectos** (*AOD*/*AOP*) es un paradigma cuyo objetivo es incrementar la modularidad (**ortogonalidad**) de los componentes mediante la separación de aspectos **transversales** (*cross-cutting concerns*).



Terminología:

- **aspect** = modularización de un aspecto de interés (*concern*) que afecta a varias clases o módulos
- **joinpoint** = especificación declarativa de un punto en la ejecución de un programa (por ejemplo, la ejecución de un método, el manejo de una excepción, etc.)
- **advice** = acción a tomar por la especificación de un aspecto dado en un determinado *joinpoint*
- **pointcut** = predicado que define cuándo se aplica un *advice* de un aspecto en un *jointpoint* determinado. Se asocia un *advice* con la expresión de un *pointcut* y se ejecuta el *advice* en todos los *joinpoint* que cumplan la expresión del *pointcut*.

Ejemplo: Shapes versión 2 (misma versión que en OCP), pero con aspectos

```
// Ficheros <X>ToString.aj (uno por aspecto)
package shapes.tostring; // para todos los toString()
aspect PolygonToString {
  String Polygon.toString() {
    StringBuffer buff = new StringBuffer();
    buff.append(getClass().getName());
     //... añadir nombre y área...
     //... añadir cada línea desde un vértice al siguiente
    return buff.toString();
  }
}
aspect CircleToString {
  String Circle.toString() {...}
}
aspect EllipseToString {
  String Ellipse.toString() {...}
}

// Drawable.java
package drawing;
interface Drawable {
  void draw();
}
```

```
// Ficheros Drawable<X>.aj
package shapes.drawing; // para todos los draw()...
import drawing.Drawable;
abstract aspect DrawableShape {
  declare parents: Shape implements Drawable;
  void Shape.draw () //template method
  {
    String drawCommand = makeDrawCommand();
    // enviar orden al motor gráfico...
  }
  String Shape.makeDrawCommand() {
    return getClass().getName() + "\n" + makeDetails("\t");
  }
  abstract String Shape.makeDetails (String indent);
}
aspect DrawablePolygon extends DrawableShape {
  String Polygon.makeDetails (String indent){...}
}
aspect DrawableCircle extends DrawableShape {
  String Circle.makeDetails (String indent){...}
}
aspect DrawableEllipse extends DrawableShape {
  String Ellipse.makeDetails (String indent){...} }
```

LSP: *Liskov Substitution Principle*

Principio de sustitución de Liskov

> Un subtipo debe poder ser sustituible por sus tipos base
>
> —Barbara Liskov,

Si una función f depende de una clase base B y hay una D derivada de B, las instancias de D no deben alterar el comportamiento definido por B de modo que f deje de funcionar

Ejemplo: Shapes versión 3

```
struct Point {double x, y;}
public enum ShapeType {square, circle};

public class Shape {
  private ShapeType type;
  public Shape(ShapeType t){type = t;}
  public static void DrawShape(Shape s) {
    if(s.type == ShapeType.square)
      (s as Square).Draw();
    else if(s.type == ShapeType.circle)
      (s as Circle).Draw();
  }
}
```

```
public class Circle: Shape {
  private Point center;
  private double radius;

  public Circle(): base(ShapeType.circle) {}
  public void Draw() {/* draws the circle */}
}

public class Square: Shape {
  private Point topLeft;
  private double side;
  public Square(): base(ShapeType.square) {}
  public void Draw() {/* draws the square */}
}
```

Problemas:

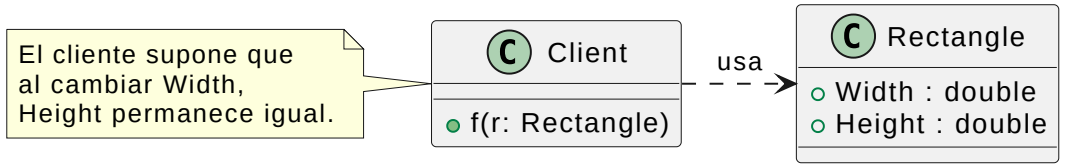
- `DrawShape` viola claramente el OCP
- Además `Square` y `Circle` no son sustuibles por `Shape`: no redefinen ninguna función de `Shape`, sino que añaden `Draw()` (violación del LSP)
- Esta violación de LSP es la que provoca la violación de OCP en `DrawShape`

A continuación, una violación más sutil del LSP...

Ejemplo: Rectángulos versión 1

De momento solo necesitamos rectángulos y escribimos esta versión:

```
public class Rectangle {
  private Point topLeft;
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.topLeft = default(Point);
    this.width = width;
    this.height = height;
  }

  public double Width {
    get { return width; }
    set { width = value; }
  }

  public double Height {
    get { return height; }
    set { height = value; }
  }
}
```



Pero un día hace falta manejar cuadrados además de rectángulos.

Geométricamente, un cuadrado es un rectángulo, así que utilizamos una relación **es-un**:

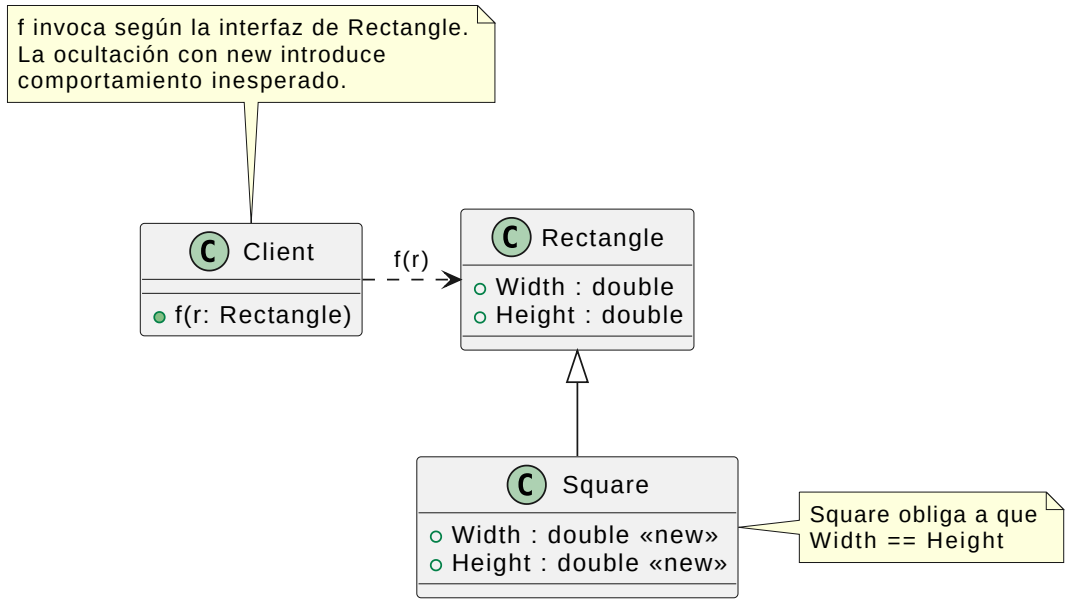
```
public class Square: Rectangle {
    ...
}
```

Problema: cuadrados como rectángulos

- Un cuadrado podría ser matemáticamente un rectángulo, pero definitivamente un objeto `Square` **no es un** objeto `Rectangle`
- Un `Square` no tiene propiedades `height` y `width`. Pero supongamos que no nos importa el desperdicio de memoria.
- `Square` heredará los métodos accesores de `Rectangle`.
- Así que hacemos lo siguiente...

Ejemplo: rectángulos versión 2

```
public class Square: Rectangle {
  public new double Width
  {
    set {
      base.Width = value;
      base.Height = value;
    }
  }
  public new double Height
  {
    set {
      base.Height = value;
      base.Width = value;
    }
  }
}
```



Ver `new` y `override` en C#

- El comportamiento de un objeto `Square` no es consistente con el de un objeto `Rectangle`:

```
Square s = new Square();
s.Width = 1;   // fija ambos
s.Height = 2;  // fija ambos

void f(Rectangle r)
{
  r.Width = 3; // calls Rectangle.SetWidth
}
```

- ¿Qué sucede si pasamos un `Square` a la función `f`?

  ¡No cambia `Height`!

- Podría argumentarse que el error era que los métodos `Width` y `Height` no se declararon `virtual` en `Rectangle`.

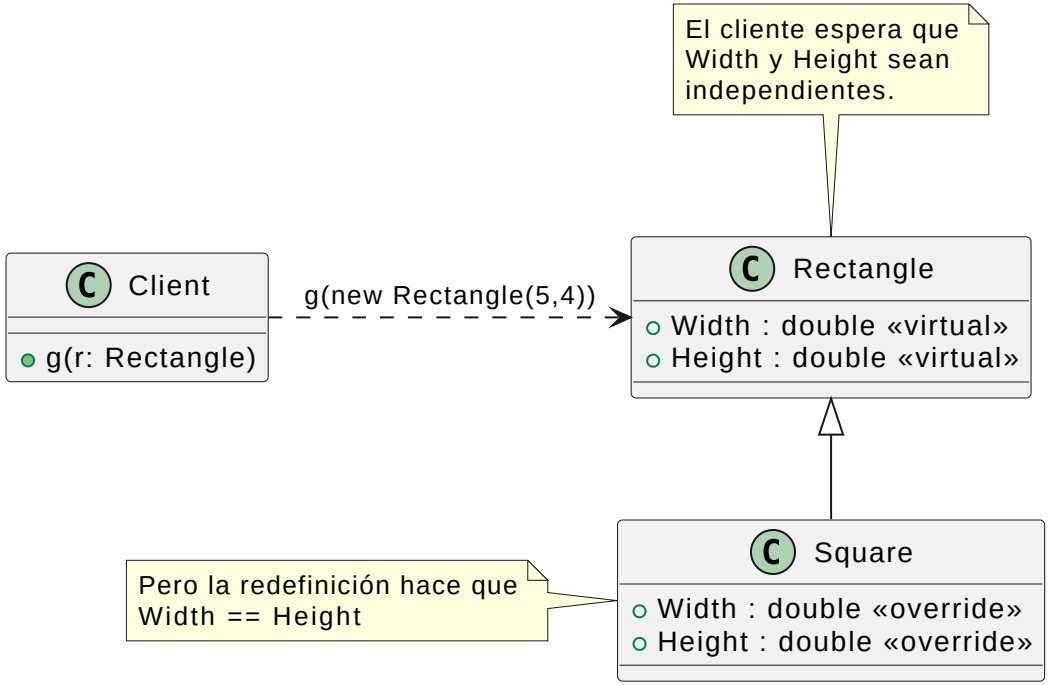
Ejemplo: rectángulos versión 3

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;
  public Rectangle(double width, double height) {
    this.topLeft = default(Point);
    this.width = width;
    this.height = height;
  }
  public virtual double Width
  {
    get { return width; }
    set { width = value; }
  }
  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}
```

```
public class Square: Rectangle
{
  public override double Width
  {
    set {
      base.Width = value;
      base.Height = value;
    }
  }
  public override double Height
  {
    set {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

Ver redefinición con `virtual` en C#

Incumplimiento del contrato



Extensión y ocultación de métodos

- Diferencia entre `new` y `override`: `new` oculta la implementación de la clase base y `override` la redefine.
- Cuando una clase derivada realiza cambios en la clase base, es síntoma de un **mal diseño**.

El LSP evidencia que la relación **es-un** tiene que ver con el comportamiento público extrínseco, del que los clientes dependen.

Ahora parece que funcionan `Square` y `Rectangle`, que matemáticamente quedan bien definidos.

Pero consideremos esto:

```
void g(Rectangle r)
{
  r.Width = 5;     // cree que es un Rectangle
  r.Height = 4;    // cree que es un Rectangle
  if(r.Area() != 20)
    throw new Exception("Bad area!");
}
```

Pero si llamamos a...

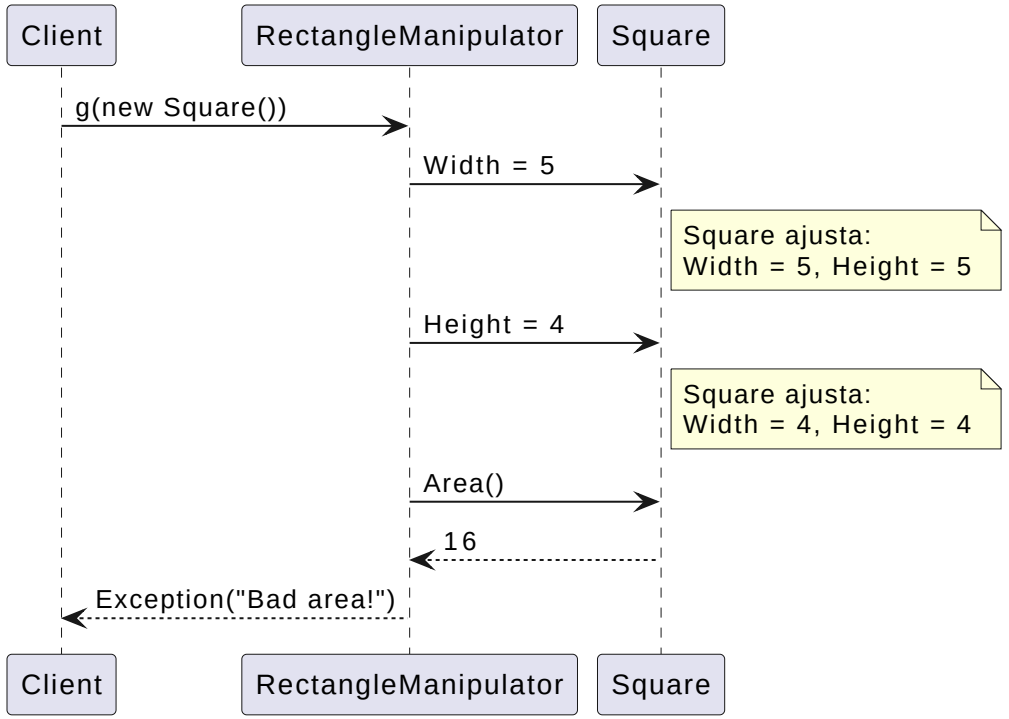
```
g(new Square())
```

... el autor de `g` asume que cambiar el ancho de un rectángulo deja intacto el alto.

Si pasamos un cuadrado esto no es así.

Violación de LSP: Si pasamos una instancia de una clase derivada (`Square`), se altera el comportamiento definido por la clase base (`Rectangle`), de forma que `g` deja de funcionar correctamente.

Secuencia del fallo



¿Quién tiene la culpa?

- ¿El autor de `g` por asumir que "en un rectángulo su ancho y alto son independientes" (*invariante*)?
- ¿El autor de `Square` por violar el invariante?
- ¿De qué clase se ha violado el invariante? ¡De `Rectangle` y no de `Square`!

Para evaluar si un diseño es apropiado, no se debe tener en cuenta la solución por sí sola, sino en términos de los *supuestos razonables* que hagan los usuarios del diseño.

Ejercicios de LSP

- Robert C. Martin & Micah Martin: Agile Principles, Patterns and Practices in C#, Prentice Hall, 2006
- Ejemplo de violación de LSP en frameworks de videojuegos: Las interfaces HasPhysics, Collidable, Controllable incluyen una dependencia (por herencia) hacia Renderable, que no siempre es cierta (por ejemplo, para un objeto invisible). Solución: arquitectura Entity-Component-System.

Diseño por Contrato

Relación entre LSP y el ***Design-By-Contract*** (DBC) de *Bertrand Meyer*:

> A routine redeclaration [in a derivative] may only replace the original precondition by one equal or weaker, and the original post-condition by one equal or stronger
>
> — —B. Meyer

- Métodos de clase declaran *precondiciones* y *postcondiciones* al redefinir una operación en una subclase derivada
  - las **precondiciones** sólo pueden sustituirse por otras más débiles/laxas
  - las **postcondiciones** sólo pueden sustituirse por otras más fuertes/estrictas

Ejemplo: rectángulos

- Postcondición del *setter* de `Rectangle.Width`
  (En C++ sería `Rectangle::SetWidth(double w)`):
  `assert((Width == w) && (Height == old.Height));`
- Postcondición del setter de `Square.Witdh`
  (En C++ sería `Square::SetWidth(double w)`):
  `assert(Width==w);`
- La postcondición de `Square::SetWidth(double w)` viola el contrato de la clase base porque es más débil que la de `Rectangle`

DIP: *Dependency Inversion Principle*

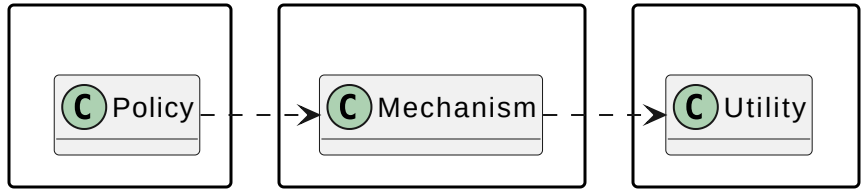
Principio de Inversión de Dependencias

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles, sino los detalles de las abstracciones

> Depend on abstractions
>
> — Robert C. Martin

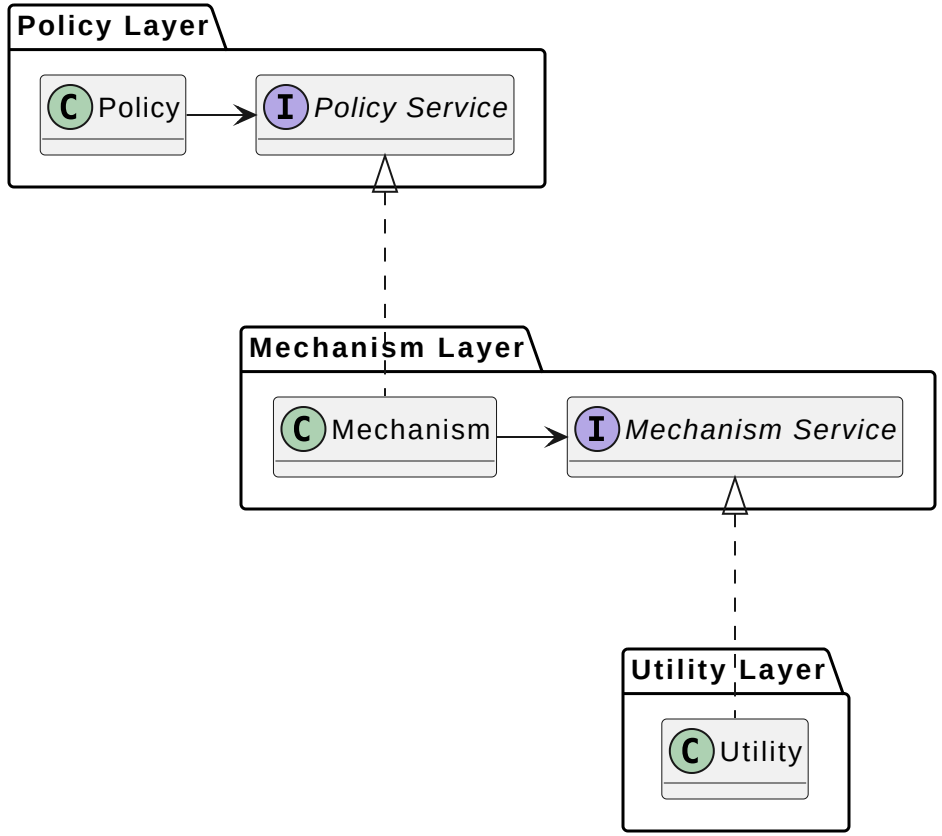
Ejemplo: estructura en capas

Diseño inicial:



- Las dependencias son transitivas
- *Policy* depende de todo lo que depende *Mechanism*.

Diseño invertido:



- Cada nivel declara una interfaz para lo que necesita de otros niveles inferiores
- Los niveles inferiores dependen de interfaces definidas en los superiores
- El cliente puede definir la abstracción que necesita (ISP)
- Cada nivel es intercambiable por un sustituto

Heurística para construcción

- Ninguna variable debería guardar una referencia a una clase concreta
- Ninguna clase debería ser derivada de una clase concreta
- Ningún método debería redefinir un método ya implementado de ninguna de sus clases base

Hay que violar alguna vez estas heurísticas, pues alguien tiene que crear las instancias de las clases concretas. El módulo que lo haga presentará una dependencia de dichas clases concretas (**inyección de dependencias**).

Gracias a la **introspección** o la carga dinámica de clases, en algunos lenguajes de programación se puede indicar el nombre de la clase a instanciar (por ejemplo, en un fichero de configuración XML o JSON).